

1N-54
1645
P303

Army-NASA Aircrew/Aircraft Integration Program (A³I) Software Detailed Design Document: Phase III

Carolyn Banda
Alex Chiu
Gretchen Helms
TehMing Hsieh
Andrew Lui
Jerry Murray
Renuka Shankar

(NASA-CR-177557) ARMY-NASA AIRCREW/AIRCRAFT
INTEGRATION PROGRAM (A³I) SOFTWARE DETAILED
DESIGN DOCUMENT, PHASE 3 (Sterling Federal
Systems) 303 p

CSCL 05H

N91-19714

Unclas

G3/54 0001645

CONTRACT NAS2-11555
June 1990



National Aeronautics and
Space Administration

Army-NASA Aircrew/Aircraft Integration Program (A³I) Software Detailed Design Document: Phase III

Carolyn Banda
Alex Chiu
Gretchen Helms
TehMing Hsieh
Andrew Lui
Jerry Murray
Renuka Shankar

Sterling Federal Systems, Inc.
Palo Alto, California

Prepared for
Ames Research Center
CONTRACT NAS2-11555
June 1990



National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

Table of Contents

1. INTRODUCTION.....	1
1.1 Identification.....	1
1.2 Scope.....	1
1.3 Purpose.....	1
2. RELATED DOCUMENTS.....	2
2.1 Applicable Documents.....	2
2.2 Information Documents.....	2
3. REQUIREMENTS & DESIGN APPROACH.....	2
3.1 Requirements and Rationale.....	3
3.1.1 Integrated Modelling Environment.....	3
3.1.2 Analysis and Decision Aiding.....	4
3.1.3 Visualization and User Interfaces.....	5
3.1.4 Incremental Development.....	6
3.2 Hardware Environment.....	7
3.2.1 Symbolics Lisp Machines.....	8
3.2.2 Silicon Graphics Computers.....	8
3.2.3 Other Processors.....	10
3.2.4 Networking Hardware.....	10
3.2.5 Peripherals.....	10
3.3 Software Environment.....	10
3.3.1 Rapid Prototyping.....	10
3.3.2 Object-Oriented Programming.....	10
3.3.3 Source Code Control.....	10
4. PHASE III DETAILED DESIGN.....	12
4.1 Introduction.....	12
4.1.1 Symbolic Modelling CSCI.....	12
4.1.2 Graphic Views CSCI.....	13
4.1.3 Cockpit Design Editor (CDE) CSCI.....	13
4.1.4 Anthropometric Model or JACK CSCI.....	13
4.1.5 Aerodynamics & Guidance Model (AGM) CSCI.....	14
4.1.6 Communications CSCI.....	14
4.1.7 Training Assessment CSCI.....	14
4.1.8 Scheduler.....	14
4.2 Demonstration Scenario.....	15
4.3 Programmatic Information.....	16
4.3.1 Risks.....	16
4.3.2 Summary of Results.....	16
4.3 Limitations.....	17
4.4 Future Directions.....	18
5. HISTORICAL INFORMATION.....	19
5.1 Phase I Development.....	19
5.1.1 Requirements and Design Approach.....	19
5.1.1.1 Summary Level.....	19
5.1.1.2 Mission Modelling.....	19
5.1.1.3 Graphics.....	19
5.1.1.4 Human Performance Modelling.....	20
5.1.1.5 Demonstration Scenario.....	20
5.1.2 Hardware Environment.....	20
5.1.2.1 Symbolics Lisp Machines.....	21
5.1.2.2 Silicon Graphics Computers.....	21
5.1.2.3 Other Processors.....	22
5.1.2.4 Networking Hardware.....	22
5.1.2.5 Peripherals.....	22
5.1.3 Software Environment.....	22
5.1.4 Programmatic Information.....	22

Table of Contents

5.1.4.1	Risks.....	22
5.1.4.2	Summary of Results.....	23
5.2	Phase II Development.....	23
5.2.1	Requirements and Design Approach.....	23
5.2.1.1	Summary Level.....	23
5.2.1.2	Modelling Environment.....	24
5.2.1.2.1	Mission Editor.....	24
5.2.1.2.2	Modeller.....	24
5.2.1.2.3	Visual Modeller.....	25
5.2.1.2.4	State Display Editor.....	25
5.2.1.3	Pilot Models.....	25
5.2.1.3.1	Anthropometric Model.....	25
5.2.1.3.2	Loading Model.....	25
5.2.1.4	Vehicle/Systems Models.....	26
5.2.1.4.1	Dynamics and Guidance Models.....	26
5.2.1.4.2	Cockpit Display Editor.....	26
5.2.1.5	World Models.....	26
5.2.1.5.1	World Models.....	27
5.2.1.5.2	Views.....	27
5.2.1.6	Analysis and Decision Aiding.....	27
5.2.1.6.1	Training Requirements Prediction.....	27
5.2.1.7	Demonstration Scenario.....	27
5.2.2	Hardware Environment.....	27
5.2.2.1	Symbolics Lisp Machines.....	28
5.2.2.2	Silicon Graphics Computers.....	29
5.2.2.3	Other Processors.....	29
5.2.2.4	Networking Hardware.....	29
5.2.2.5	Peripherals.....	29
5.2.3	Software Environment.....	29
5.2.4	Programmatic Information.....	30
5.2.4.1	Risks.....	30
5.2.4.2	Summary of Results.....	31
6.	APPENDICES.....	31
6.1	Glossary, Definitions, Abbreviations.....	31
6.	ANNEXES.....	33
Annex A	— Symbolic Modelling CSCI.....	33
Annex B	— Views CSCI.....	33
Annex C	— Cockpit Design Editor CSCI.....	33
Annex D	— Anthropometric Model "JACK" CSCI.....	33
Annex E	— Aerodynamics/Guidance CSCI.....	33
Annex F	— Communications CSCI.....	33
Annex G	— Training Assessment CSCI.....	33

1. INTRODUCTION

1.1 Identification

This document establishes the requirements and detailed design of the Army-NASA Aircrew/Aircraft Integration (A³I) Computer Program System (CPS) and subordinate Computer Software Configuration Items (CSCI). Introductory descriptions of the processing requirements, hardware/software environment, structure, I/O, and control are described for the overall CPS, with detailed discussion of the individual CSCIs included in Annexes A-F. Development history and rationale for the A³I CPS is also provided, although the emphasis is on describing its functionality at the end of Phase III.

1.2 Scope

This document is intended for the use of programmers and other technical specialists working on the development of the prototype A³I computer-aided engineering workstation. Sufficient high level information is provided to allow any reader to become familiar with the objectives of the A³I Program, the current (as well as previous) overall architecture and development philosophies, while at the same time containing implementation detail useful to programmers involved with specific application software modules.

This document is revised during each phase of development, tracking the history of changes to the configuration, and more importantly, the motive for such changes. Familiarity with previous phases of development is assumed. For historical information regarding Phases I and II, the reader is urged to consult Section 5, Historical Information or the reference documents from previous phases.

1.3 Purpose

The purpose of the A³I CPS is to serve as the integration and demonstration framework in which the applied research products of the Program are instantiated. This framework is a prototype computer-aided engineering workstation suite termed MIDAS—for Man-machine Integration Design and Analysis System—that integrates human factors engineering with other vehicle/system design disciplines at an early stage in the development process of manned vehicles.

MIDAS is intended to be a human factors engineering tool which assists design engineers in the conceptual phase of rotorcraft crewstation development and helps anticipate crew training requirements. The system provides designers with interactive, analytic, and graphical tools which permit early integration and visualization of human engineering principles.

70 to 80 percent of the life-cycle cost of an aircraft is determined in the conceptual design phase. After hardware is built, mistakes are hard to correct and concepts are difficult to modify. Engineers responsible for developing crew training simulators and instructional systems currently begin work after the cockpit is built and too late to impact its design. When complete, MIDAS will give designers an opportunity to "see it before they build it," to ask "what if" questions about all aspects of crew performance, including training, and to correct problems early. The system is focused on helicopters, but is generic and permits generalization to other vehicles.

MIDAS is similar in concept to computational tools such as finite element stress analysis and computational fluid dynamics which are used to improve designs and reduce costs. The results of the computational analysis are presented visually. The workstation uses models of human performance and a computational simulation of "manned flight" to evaluate the cockpit design. The results are presented graphically and visually to the design engineers, often as a computer animation of manned flight.

The components of MIDAS at the end of Phase III include:

- 1) The Symbolic Modelling CSCI, containing methods to represent and decompose the design of a crew station, the required mission, the environment, human performance models of the crew, and the causal relations existing between these elements for analysis. A tick-based simulation is "driven" by the interactions of these models/activities.
- 2) The Views CSCI which is used to create and observe, from several perspectives, the 3-D graphic environment of the mission simulation.
- 3) The Cockpit Design Editor or CDE CSCI which contains 3-D CAD utilities for designing the cockpit geometry, instruments, controls, and displays.
- 4) A 3-D, interactive, anthropometric pilot model or graphic mannequin called the JACK CSCI.
- 5) The helicopter Aerodynamics and Guidance CSCI.
- 6) The Data Communications CSCI, facilitating the intermachine communications and dynamic message sharing between the above components.
- 7) The Training Assessment CSCI, providing means to estimate the training media, instructional techniques, and time necessary to qualify in the cockpit under development.

The A³I program began in fall 1984 and has completed three major phases of development toward a 1994 target date for a full prototype system. Phase III was focused on the expansion of several of the aforementioned components, particularly with an emphasis to make explicit their sensitivity to changes in cockpit design.

2. RELATED DOCUMENTS

2.1 Applicable Documents

A³I Executive Summary, 15 April 1986

2.2 Information Documents

A³I Phase II Human Factors/Computer-Aided Engineering Workstation Suite Architecture Description Document, Revision 1, 30 Nov 87

Human Performance Models for Computer-Aided Engineering, National Research Council Committee on Human Factors, National Academy Press, 1989

3. REQUIREMENTS & DESIGN APPROACH

3.1 Requirements and Rationale

Because they are central requirements for the overall A³I Program, a number of key developmental aspects must be mentioned prior to describing the MIDAS computational environment or Phase III specifics in paragraphs 3.2, 3.3, and 4, respectively.

First, the tools and models of MIDAS are intended to support three major phases of the design process. These include specification, static analysis, and dynamic analysis. Tools such as the Cockpit Design Editor and portions of the Symbolic Modeling CSCI exist to allow the user to input or "specify" the elements of the mission, crew, cockpit, and environment that are given or known. Secondly, a number of the tools and models, such as the anthropometric model or Jack CSCI, can be used for static analysis. Here, the intent is to support analyses such as reach or visibility which can be answered for static or fixed conditions and do not require complex interactions that can only be discovered through a dynamic scenario. Finally, components such as the Aerodynamics & Guidance Model, the Communications CSCI, and portions of the Symbolic Modeling CSCI exist to aid in dynamic analysis through a model-based simulation of the operator, equipment, and environment interactions. This simulation uses a discrete time or tick-based approach to explore the changing world, operator, and equipment states and propagate their effects over time. Since computational models and principles of human performance replace the operator found in typical man-in-the-loop simulations, MIDAS contains no requirement for real-time human interaction—a feature which would be unnecessarily restrictive for our objectives.

As an important final note, the A³I Program attempts to achieve these design support goals by using extant software and systems to the greatest extent possible. A considerable amount of research effort, money, and time has been expended nationally to produce analytic methods, models and structures representing the behavior and functions of human operators, avionics systems, and missions, with varying degrees of success. Where possible, the staff will selectively employ those models/tools which have already been developed, engaging in research and development for such components only when a critical void is encountered. The advice of the National Research Council's Committee on Human Factors (study group on Human Performance Models and their reference report) has been solicited on this matter, resulting in our adoption of the following guidelines.

3.1.1 Integrated Modelling Environment

An enormous amount of modelling is anticipated throughout the life of the Program, with the operator, helicopter, and world as the major categories of models. Since the effort envisioned is primarily one of integrating the many and diverse software modules within a cohesive, inspectable workstation suite, several methodologies have been used to enhance this process. The A³I Program has elected a simulation-based approach to system design and evaluation that attempts to make extensive use of graphic and iconic representations of the underlying model structures. Models are generally prescriptive, providing results relevant to mission success in terms of such parameters as performance, errors, duration, and rates. Although the use of extant, validated models is preferred, it will often be the case, (particularly in the domain of human modelling) that relatively unproven, recently developed research results may be incorporated in the workstation for evaluation. It is intended that the MIDAS workstation will serve a dual purpose in this role by providing an integrated environment for model inspection and testing. Figure 1 provides a notional view of the range and types of model interaction anticipated within the MIDAS environment.

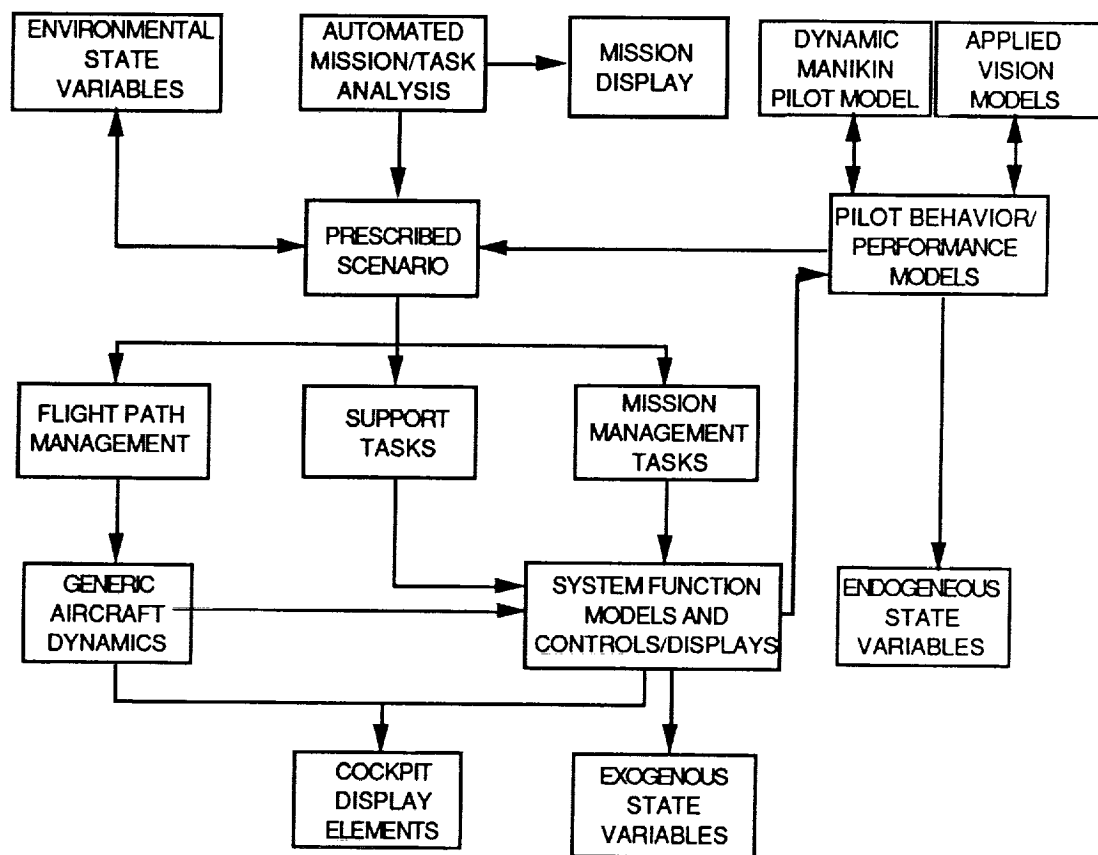


Figure 1. Notional MIDAS Workstation Models & Integration

The pilot is the critical link in the system being represented and the specifics of performance must ultimately determine overall behavior of the system being designed. At each step in the selected scenario, and for the specific control/display design being investigated, the pilot performance and behavioral models provide appropriate input to the aircraft dynamics models, equipment models and environment to generate the next event-step in the scenario as it unfolds. Thus if mission demands, display interpretation requirements, information handling demands, or multiple tasking, etc. exceed reasonable pilot capabilities, the operator performance and behavior models should ideally supply the kind of reduced-performance response which should be expected.

The use of abstraction is also strongly encouraged as part of this integrated environment. The goal is to be able to simplify each model from precise computational representations to approximate, qualitative models, as determined by the interactions under study and the answers desired. As the workstation architecture evolves, it is expected that both deterministic and stochastic constructs will be required within the same simulation environment to make use of the widest possible range of extant models.

3.1.2 Analysis and Decision Aiding

Analytical capabilities contained within MIDAS are intended to emphasize three aspects of the simulated operator's behavior—timeliness, accuracy, and loading—along with their contributions to and results from man/machine system performance, equipment design, and

training. Because the workstation focus to date has been on model development and integration, the majority of the analysis tools envisioned are still undeveloped. Currently, task/loading timelines are the primary means of analysis. This core capability is augmented with prototype examples of how human engineering data compendiums and training analysis tools can also aid in the design process.

As previously mentioned, both static and dynamic analysis procedures are required within MIDAS. For example, reach and fit analyses for cockpit controls can be performed on geometric representations of the cockpit without the need for elaborate vehicle dynamics/systems or human performance simulation models. On the other hand, modelling the complex interactions of competing tasks during the performance of some mission requires considerable dynamic modelling of both human behavior/performance and vehicle systems to capture the context-sensitive, time-varying nature of task sequencing and resultant loads. The key is to recognize that within the workstation no single figure of merit or analysis tool will provide all of the answers required in man-machine analysis.

Consequently, as dictated by the capabilities of the underlying models, analysis capabilities will be continuously added and refined. MIDAS capabilities are not intended to obviate the designer's job or stifle creativity. Interpreting "grey" or subjective results, as well as the aggregation of many different findings into a summary figure, are tasks which are best left to human designer/analyst judgement. Instead, MIDAS is concentrated on providing analysis within a decision aiding context by helping designers determine the appropriate remedial action in the event of failure to perform a mission, excessive error rates, overloads or exceeding other parameters deemed significant by the user. When black-and-white answers can be determined (such as whether an instrument can be seen or reached) a black-and-white result will be given. However, most of MIDAS's use will come from iterative processes where the designer can vary conditions and model parameters until they feel adequate data is available. Many times, this information will be qualitative in nature, leading not to specific conclusions, but descriptive enough that when combined with human expertise, the appropriate evaluations can be made.

The substantial body of data and information generated during the simulation must be interpreted in a manner that is meaningful and relevant to various specialists evaluating a candidate cockpit design. To achieve such goals, graphic and analog displays of chosen parameters of interest is encouraged. This topic is further described below.

3.1.3 Visualization and User Interfaces

The Program's emphasis on visualizing the results of complex interactions is intended to facilitate the use of MIDAS in a design/analysis session without requiring an undue amount of knowledge about the underlying implementation.

Data and information selected as interesting is required to be presented in a form the designer/user finds easy to interpret. Alphanumeric tabular forms, though easiest to generate, are often ill-suited to designer/user needs, hence alternate forms are required to facilitate the designer/user's easy insight into the overall progress of the simulation and a global understanding of complex and interrelated man-machine factors. The use of graphic and iconic representations also facilitates communication between designers from different technical disciplines by substituting commonly-understood pictures for words which may have different meanings to each. The result is a depiction of the impact of design decisions in a form which is meaningful to a wider range of potential users.

The user's interface to the computational tools and models of MIDAS is extremely important. However, since the Program's charter is to develop prototype facilities on an architecture that is continually evolving and may or not be common to prospective delivery

platforms, a polished, consistent interface across all of the applications has not been a priority. Furthermore, the reliance on outside universities and research organizations for various fundamental workstation components makes a unified interface concept difficult. As a general requirement however, each application developer is encouraged to use cognitive or computer science research findings in human-machine interactions when available. Pop-up windows, pull-down menus, and the manipulation of graphic or iconic representations, similar to those made popular in the Apple Macintosh® desktop, have migrated into the majority of the CSCIs. The sophistication of the individual interfaces is generally a function of the maturity of the underlying tool/model, with roughly 15-20% of the development effort committed to the user interaction.

In future phases, greater emphasis will be placed on the user interface area. Since cognitive or computer science research findings have typically not kept pace with emerging hardware technologies (voice I/O, touch panel overlays, 6 DOF mice, eye trackers, stereoscopic displays, etc.), the Program will initiate an effort to develop a unified, principle-based approach to user interfaces that is based on human-computer interactions (HCI) research at various universities and industry centers.

3.1.4 Incremental Development

The A³I Program's prototype MIDAS workstation is developed in phases, each adding another increment of functionality to the existing configuration. Integration of software configuration items is performed at NASA/Ames Research Center by in-house staff in cooperation with outside support as required. Upon the completion of each 8-14 month phase, demonstrations are held, feedback solicited, compiled and assimilated, and planning meetings conducted to propose appropriate work for the following phase of development. Work breakdown structures and schedules are developed to track phased development in each major work area. Documentation, primarily this Software Detailed Design Document, is developed and updated coincident with new/modified software.

Figure 2 below depicts major phase milestones since the Program's inception.

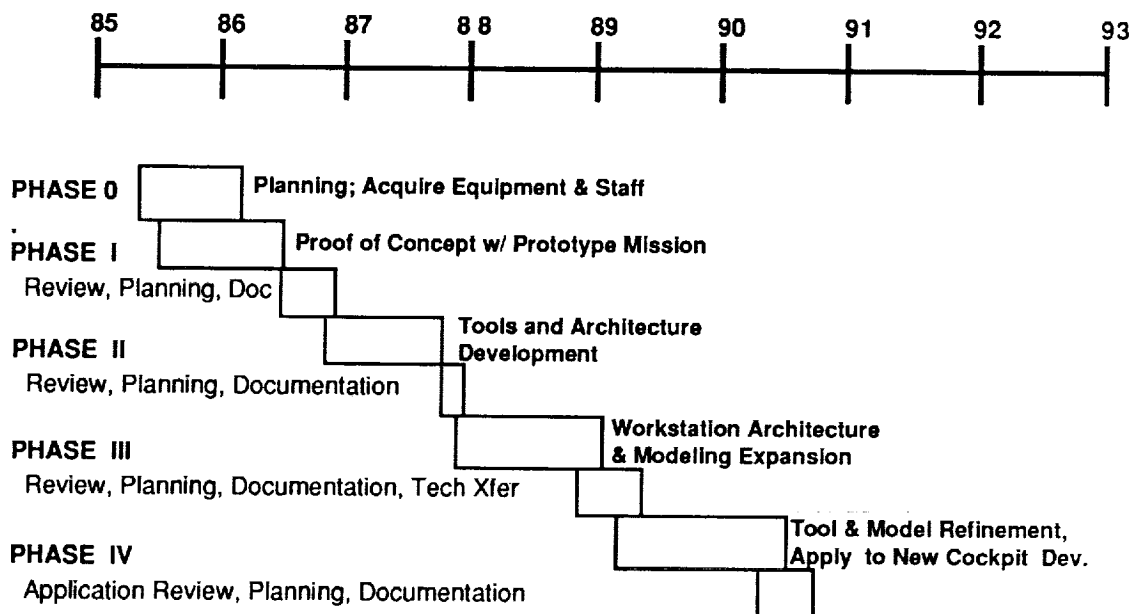


Figure 2. A³I Program Timeline

3.2 Hardware Environment

The specific computational hardware environment for each CSCI is described in detail within Annexes A-F. The progression of such hardware throughout the development phases is also described in Section 6 Historical Information. In general, the program has adopted the requirement to use existing and proven hardware—namely networked Silicon Graphics Workstations and Symbolics Lisp Computers. Beta architectures and the development of unique hardware in cases where workable, off-the-shelf solutions exist are not permitted. Exceptions to this standard in the later stages of workstation development may be investigated when performance is being optimized and potential delivery platforms explored.

The hardware architecture in place at the end of Phase III is depicted in Figure 3 below. These components, together with their resident software and peripherals are described in further detail in the subsections which follow.

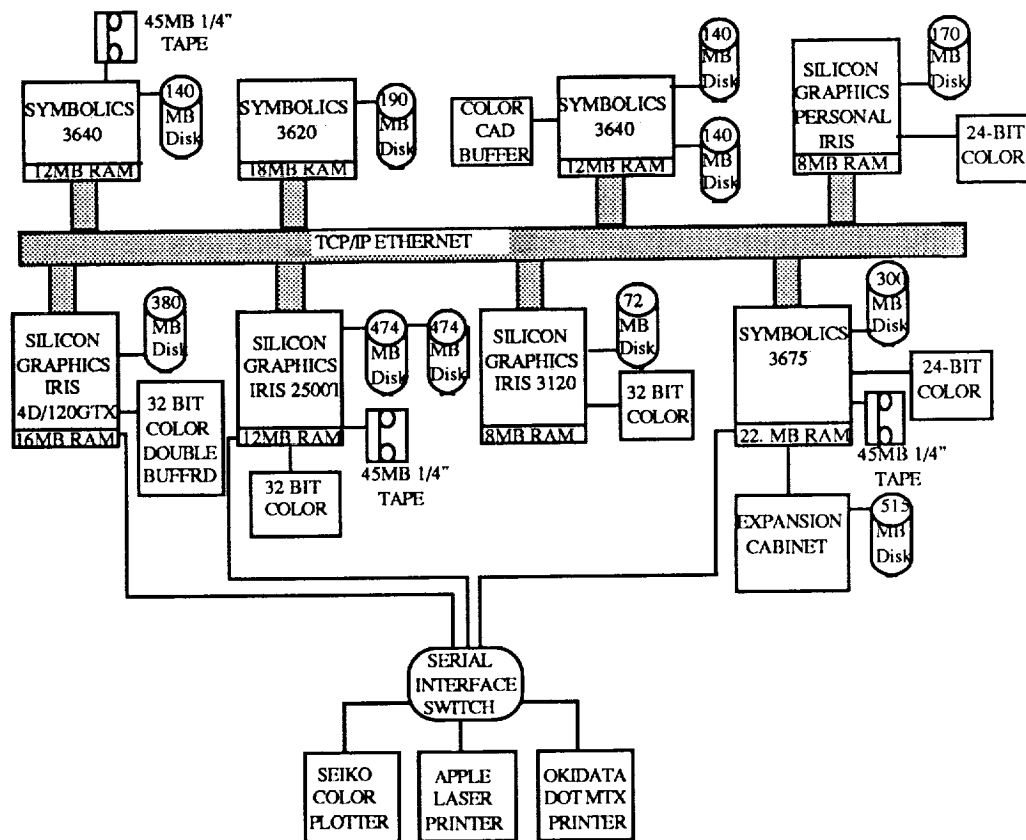


Figure 3. Phase III Hardware Configuration

Two of the Symbolics and two of the Silicon Graphics computers listed above are essentially development assets and are not required for demonstration or actual use.

3.2.1 Symbolics Lisp Machines

Model 3675 Color Workstation (Barracuda) consisting of:

- Monochrome Console with OCLI filter
- Keyboard & Mouse
- 45 MB 1/4" Cartridge Tape Drive
- Ethernet Controller and Transceiver
- 22.5 MB RAM
- Enhanced Performance Option
- 338 MB Fujitsu Eagle Disk
- 550 MB CDC Disk
- Model CG70-FB02 High Resolution, 24-bits/Pixel Color Frame Buffer
- Tektronix 19" Color RGB Monitor
- Model OP36-FPA1 Floating Point Accelerator
- Symbolics # SLAN-FORT Fortran 77 Compiler
- Symbolics # STCP-1 TCP/IP Software
- S-Group (S-Paint, S-Geometry, S-Render, S-Dynamics, and color 6.0 V405.13)
- Genera 7.2

Model 3640 Color Workstation (Puffer) consisting of:

- Monochrome Console with OCLI Filter
- Keyboard & Mouse
- 45 MB 1/4" Cartridge Tape Drive
- Ethernet Controller and Transceiver
- 11.25 MB RAM
- 2-140 MB Disks
- CAD Buffer
- Tektronix 19" Color RGB Monitor
- Symbolics # SLAN-FORT Fortran 77 Compiler
- Symbolics # STCP-1 TCP/IP Software
- S-Group (S-Paint, S-Geometry, S-Render, S-Dynamics, and color 6.0 V405.13)
- Genera 7.2

Model 3640 Monochrome Workstation (Squid) consisting of:

- Monochrome Console with OCLI Filter
- Keyboard & Mouse
- Ethernet Controller and Transceiver
- 13.5 MB RAM
- 2-140 MB Disks
- Symbolics # SLAN-FORT Fortran 77 Compiler
- Symbolics # STCP-1 TCP/IP Software
- Genera 7.2
- Automated Reasoning Tool (ART) Version 3.2

Model 3620 Monochrome Workstation (Sea Slug) consisting of:

- Monochrome Console with OCLI Filter
- Keyboard & Mouse
- Ethernet Controller and Transceiver
- 18 MB RAM
- 190 MB ST506 Disk
- Symbolics # SLAN-FORT Fortran 77 Compiler
- Symbolics # STCP-1 TCP/IP Software
- Genera 7.2

3.2.2 Silicon Graphics Computers

W-2500A Workstation (Orca) consisting of:

- 19" High Resolution Monitor
- Keyboard & Mouse

45 MB 1/4" Cartridge Tape Drive
Ethernet Controller and Transceiver
12 MB RAM
2 474 MB Fujitsu 10.5" Disk Drives
HU-T04 Turbo Option W/4MB RAM
H3-FPA Floating Point Accelerator
H-DM4A 1024x1024x4 Display Memory
H-ZC2 Z Clipping Assy
C-WTCP IP/TCP Software
P-DBX Dial/Button Box
Unix System V with BSD 4.2
NFS
C Compiler
IRIS Graphics Library II and Window Manager

W-3120 Workstation (Manta) consisting of:

19" High Resolution Monitor
Keyboard & Mouse
Ethernet Controller and Transceiver
8 MB RAM
72 MB Winchester Disk Drive
H3-FPA Floating Point Accelerator
H-DM4A 1024x1024x4 Display Memory
H-ZC2 Z Clipping Assy
C-WTCP IP/TCP Software
P-DBX Dial/Button Box
Unix System V with BSD 4.2
NFS
C Compiler
IRIS Graphics Library II and Window Manager

W-4D120GTX PowerSeries Workstation (Coral) consisting of:

19" High Resolution Monitor
Keyboard & Mouse
Ethernet Controller and Transceiver
16 MB RAM
380 MB ESDI Winchester Disk Drive
Double Buffered 1280x1024x4 Display Memory
Double Buffered Alpha
24 bit Z buffer
C-WTCP IP/TCP Software
P-DBX Dial/Button Box
IRIX System V release 4D1-3.1D
NFS
C Compiler
C++ Translator
Fortran 77 Compiler
IRIS Graphics Library II and 4Sight Windowing System

W-4D20G Personal IRIS Workstation (Urchin) consisting of:

19" High Resolution Monitor
Keyboard & Mouse
Ethernet Controller and Transceiver
8 MB RAM
170 MB SCSI Winchester Disk Drive
1280x1024x4 Display Memory
Double Buffered Alpha
C-WTCP IP/TCP Software
IRIX System V release 4D1-3.1D
NFS
C Compiler
IRIS Graphics Library II, 4Sight Windowing System, and Environment Manager

3.2.3 Other Processors

An unused DEC MicroVax II is on loan to a neighboring branch.

3.2.4 Networking Hardware

CableTron MT-800 Ethernet/IEEE 802.3 Transceiver

3.2.5 Peripherals

Okidata Model 2410 Dot Matrix Printer

Apple LaserWriter Plus Laser Printer

Seiko Instruments D-Scan CH5312 Color Printer & Multiplexor

GraphOn GQ-250 ASCII Terminal & Keyboard (2)

Hewlett-Packard HP 700/22 ASCII Terminal & Keyboard (3)

3.3 Software Environment

A powerful software development and integration environment is obviously essential to A³I's progress with human engineering tools and models. However, as stated previously, it is not the charter of the A³I Program to develop new models. Rather, the Program will produce a prototype framework for the integration and evaluation of those that already exist, developing new models only when necessary. These models exist in various languages (Fortran, C, Pascal, Lisp, Prolog, etc.), simulation views (continuous, discrete, network, combined) and paradigms (dataflow, functional, procedural, descriptive). Consequently, one of the Program's central challenges is to design and implement a general environment that can accommodate these variations easily and quickly while maintaining inspectability and modularity at various levels of modelling abstraction. Three key development philosophies support these goals:

3.3.1 Rapid Prototyping

Since its inception, the A³I Program has emphasized the evaluation of architectural issues and model requirements through rapid prototyping. Much of what is being attempted by the software development staff is radically new with no known, proven methods. Rapid prototyping as a means to develop the various tools and models has both guided and facilitated subsequent effort, providing the opportunity for scientific scrutiny and using-community feedback before an overwhelming development effort is expended.

3.3.2 Object-Oriented Programming

The A³I Program has adopted the use of object-oriented programming methods, where practical, in an effort to manage the complexity of the simulation software through modularity, abstraction, as well as promote graceful incremental software development. While not universally applied to every CSCI, the object-oriented paradigm is a natural match to the structure of the man-machine integration problems investigated by A³I (helicopter, operator, and world), supports extensive reuse of software structures, and appears to be an evolving standard of the software development community.

3.3.3 Source Code Control

Software for which the Program does not have access to source code (with modification rights) is generally not permitted in the configuration. Encumbrances to releasing internally

developed software to other organizations will cripple the Program's success since technology transfer is a primary concern. However, exceptions have been allowed for packages with clearly defined and easily accessible software interfaces that would be inappropriate for the Program to develop, or software development utilities (shells, debuggers, process performance metering, function libraries, etc) that offer superior performance which other organizations have ready access to.

Generally, the graphic design and analysis tools were built using C, Unix BSD 4.3, and the IRIS Window Manager/Graphics Library II. The Multigen modelling package was used as the underpinnings for the CDE and Views components, as well as a visualization medium for the Aerodynamics and Guidance CSCI. Most Lisp tools and models were built using Symbolics Common Lisp, with the Flavors extension, under Genera 7.2. The Training Analysis CSCI uses the Automated Reasoning Tool (ART) as a shell for its inference requirements. The S-Packages were available for use in displays, although not heavily relied upon during this phase. Communication software uses a local Ethernet to enable inter-machine message passing and simulation synchronization. The distribution of the Phase III models, tools, and displays is shown in Figure 4 below.

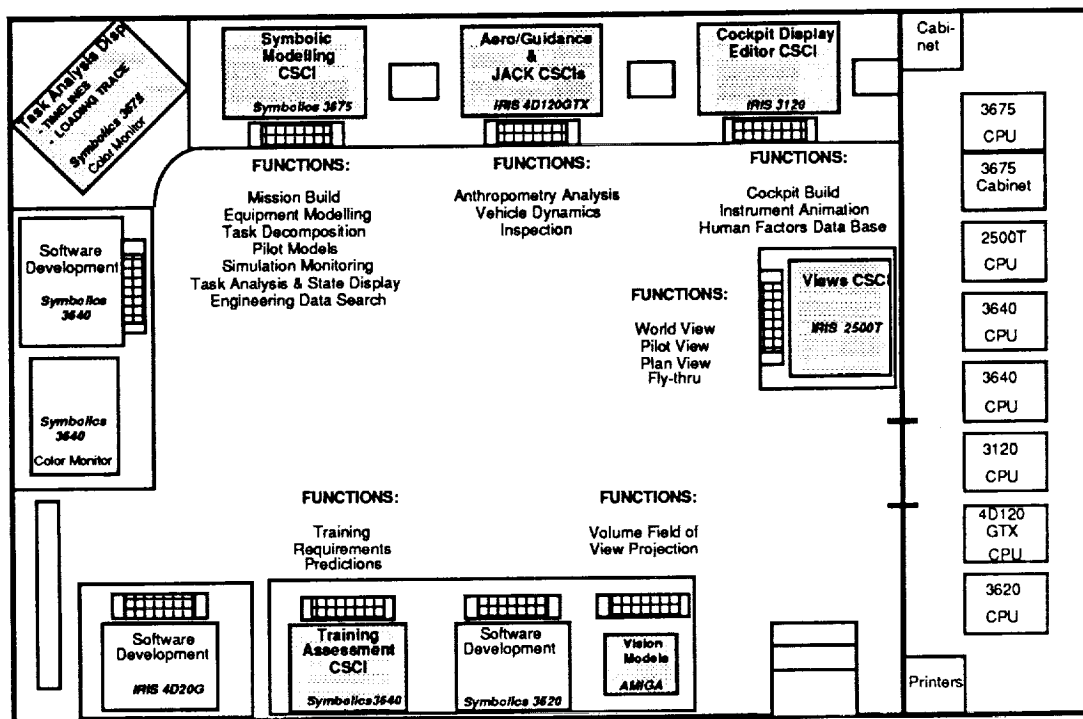


Figure 4. Distribution of Phase III Software Components and Displays within the A³I Lab

Absent from the figure above is the Communications CSCI. This component is actually distributed among all of the various Symbolics and Silicon Graphics machines as dictated by the integration requirements. Currently, the capability exists to share the following variables among the graphic and symbolic computers during a simulation.

For each Helicopter (Ownship and Wing): *Ownship only

- Helicopter position (x, y, z)
- Helicopter orientation (yaw, pitch, roll)
- Altitude (AGL), Airspeed, Groundspeed
- Velocity in each axis
- Engine torque
- *Cyclic position, pedal position, collective position

For each Truck or Ground Vehicle (up to 6):

- Truck position (x,y,z)
- Truck orientation (yaw)

For each Missile on board a Helicopter or Ground Vehicle (up to 9):

- Missile position (x, y, z)
- Missile orientation (yaw & pitch)

Additionally, each object above has several "flags" which can be set to communicate when an event such as a "hit" or an "explosion" occurs.

4. PHASE III DETAILED DESIGN

4.1 Introduction

Responses to the Phase II demonstrations, as well as discussion at the Phase III off-site reinforced the need to continue the development of the core set of A³I models and tools. However, emphasis needed to be placed on explicitly addressing how such models and tools would be sensitive to cockpit design change. Furthermore, the enhancement of the present applications and the start of new components must be anchored to a "real world" application for effective demonstration positions. The Program office decided to use a detailed, "vertical slice" of the conceptual development process as a method to illustrate the intended use of the workstation. The AH-64A mission and cockpit was the focus of the phase, with empirical flight test results from an AH-1 Cobra Communication Study as a source of specific task data. These objectives required a degree of integration and detail previously impossible, and drove a number of development requirements for the individual MIDAS components. These requirements and the resulting design approach are briefly described below with a majority of the details provided in Annexes A-F.

4.1.1 Symbolic Modelling CSCI

Consisting of a mission editor, task decomposition aids, simulation constructs and executive, as well as state displays for task analysis, the focus for the Symbolic Modelling CSCI during this phase was on the design and coding of a generalizable framework for symbolically representing the functions of cockpit equipment used to accomplish mission tasks. This framework allows various cockpit alternatives to be evaluated without completely re-editing the mission decomposition, since the design maintains a distinction between the physical structure (or state operators) of the equipment, and the functional requirements (or inferred goals) required by the task. Previous A³I symbolic models of the mission and pilot tasks failed to explicitly depict the relationship of the equipment design and the primitive task actions, loading values, or timelines. Results of the "vertical slice" example (report phase line) were used to demonstrate this process and successfully compared to actual data from a similar AH-1 flight test conducted by the Aeroflightdynamics Directorate. Task timeline, resource use, and loading displays were similar in concept to those used during previous phases.

The mission editor and equipment representation portions of this CSCI are valuable in themselves since they allow a user to rapidly explore the detailed task ramifications of various cockpit designs. However, their real strength comes when used as part of the simulation. Using an hierarchical decomposition, mission, task, environment, or operator objects are instantiated when their conditions are met, executing their assigned procedures and spawning new activities. Contained within the various task objects is information on temporal relationships, preconditions, logical constraints, loading, and any subtasks. In this manner, the decomposed mission serves as a forcing function for the various models/tools used during a simulation.

4.1.2 Graphic Views CSCI

Phase III development requirements for the Views CSCI are best described as enhancements to the well-received capabilities existing in Phase II. The internal resolution of the geometric modelling package was reduced from 3/8 inch to 1/256 inch, permitting sharper and more detailed rendering of the DMA terrain and moving models. Also added was the capability for the user to select by mouse a viewing position from anywhere within the mission gaming area. Existing low-detail helicopter models were also replaced by the fully populated AH-64A model developed using the CDE. Finally, the Views CSCI was ported to the new IRIS 4D and modified as dictated by a new version of the underlying MultiGen software.

4.1.3 Cockpit Design Editor (CDE) CSCI

Existing CDE software was ported to the IRIS 4D and improved with the addition of pop-up user windows, an hierarchical data base of instrument information and characteristics for human factors analysis, and improved mouse operations. The CDE was successfully used to build a detailed 3-D model of the complete AH-64A pilot cockpit and instrumentation, providing both a feasible application of its capabilities, as well as cockpit/craft models for the remaining workstation elements. An F-16 cockpit was also created for potential use within the fixed-wing community. A fair amount of debugging of the initial CDE code took place with these full-scale CAD attempts, along with code changes brought about by a new version of the underlying MultiGen software.

4.1.4 Anthropometric Model or JACK CSCI

Under our grant to Dr Norm Badler and the University of Pennsylvania, the 3-D dynamic anthropometric model was also ported to the IRIS 4-D and now includes fully defined body parts, limb joint constraint settings, adjustable eye viewpoints, as well as improved animation capabilities. This model was placed inside the AH-64A cockpit designed with the CDE and saved in a Psurf format, and "instructed" to perform elementary psychomotor operations through mouse operations. An somewhat unrealized goal for this phase involved "driving" Jack through the task decomposition commands on the Symbolics. Although demonstrated in a small scale, this level of integration during a simulation was not pursued due to the lack of a synchronizing "time concept" within Jack. A number of miscellaneous improvements were also added by the developer including a spread-sheet style anthropometric database with editing options, increased functionality in shaded (rather than wire-frame mode), and a new sliding window user interface. Jack was used during the phase to perform reach analyses and a simplified visual occlusion test using 5th and 95th percentile males/females in the Apache Cockpit.

Far more rigorous vision models were kicked-off this phase with a grant to the New York Association for the Blind, and a contract with the David Sarnoff Research Center. These models of volume field of view and legibility, respectively, will be integrated within the Jack environment but are not detailed within this document since preliminary versions are not expected until Phase IV.

4.1.5 Aerodynamics & Guidance Model (AGM) CSCI

The MIDAS AGM is a two-part model representing rather generic helicopter guidance and dynamics for uncoupled controls. Given the current position, orientation, and angular rates, the guidance portion of the model determines the control inputs required to fly to the next waypoint with its associated position, altitude, and airspeed. The aerodynamics portion of the model uses the computed controls to determine the helicopter's next position, orientation, etc., based on the simulation tick interval. One of the specific desires this phase was to convincingly demonstrate that A³I indeed had a viable aero model and that it could be used to provide vehicle position, orientation, and rate information to other MIDAS components. To accomplish this task, modifications were made to the existing code to use MultiGen graphic windows for displaying the helicopter's pitch, roll, and yaw characteristics, as well as icons representing the simulated cyclic, pedal, and collective inputs computed by the guidance portion of the model. The AGM CSCI was also ported from the Symbolics computer to the IRIS 4D to take advantage of the additional compute power and better Fortran environment. Finally, extensions were provided which allow the user to select waypoints for the simulation using the mouse. Integration between the AGM and Symbolic Modelling CSCI's was limited due to the effort required to accurately portray the specific piloting activities required in the demonstration scenario.

4.1.6 Communications CSCI

The IRIS to Symbolics Communications software developed during Phase II was significantly improved this phase by making it bi-directional and expanding the types of data passing supported. Previously, only integer data types were sent using one-way communications from the Symbolics to the Iris machines. During Phase III the capability to send character strings and floating point data was added, along with the bi-directional feature needed to support a higher level of integration among the MIDAS CSCI's. However, since most of the application developments demonstrated during this period were "stand-alone" oriented, the major use of this CSCI was to transmit position, orientation, rate, and limited engine parameters from the AGM to the Views/CDE CSCI's for animation.

4.1.7 Training Assessment CSCI

Training assessment was greatly augmented and refined during Phase III. During Phase II, an instructional system was assigned (by table look-up) to train each task by matching task characteristics with attributes of instructional systems within the task's learning category. The Phase III approach used ART (the Automated Reasoning Tool) and Common Lisp on the Symbolics to develop a prototype knowledge-based system. This process uses the instructional systems design (ISD) process to assign each task a set of learning experiences (such as explanation, demonstration, part-task training, and full task training) along with a medium for each learning experience (such as textbook/workbook, interactive slide/tape, lecture with visual aids, videodisc/CBT, CFT, CPT, OFT, WST without and with motion, and the actual system). For each learning experience/media assignment, a time to train is computed, based on the task, operator, and equipment characteristics. The Phase III training approach was heavily based on previous work performed by the Logicon Corporation under contract to the Air Force. Training assessment is accomplished on an individual task basis, with no attempt made to address the grouping of tasks into lessons or courses.

4.1.8 Scheduler

Work was also begun this phase on a dynamic reactive scheduling component to address the sequencing and scheduling a pilot may perform as a means to control his task performance and timeliness. While not committed to code during Phase III, significant headway into the specific objectives and approaches for this state-of-the art component were achieved. Because it was not

completed during the phase, formal documentation for the scheduler is not provided as part of this Phase III SDDD.

4.2 Demonstration Scenario

The Phase III demonstrations consisted of a 30 minute introductory briefing by the program director, followed by 1.5 hours of application demonstrations by the staff. The "vertical slice" into the development process was emphasized as the attendee was essentially "walked through" the conceptual development process for a potential communication switch change on the AH-64A. The demonstration objective was to describe the potential interactions and conflicts arising from moving the radio select button from the ICS panel on the front bulkhead to the cyclic (as was done in the AH-1 flight test). The capability for A³I to support three phases of the design process was stressed: specification, static analysis, and dynamic analysis.

Beginning with the Views CSCI, the projected DMA gaming area was portrayed as the mission environment and the inherent capabilities of visualization emphasized. Next, the mission editing component of the Symbolic Modelling CSCI was introduced, along with its facilities to input the scenario for a unmasking maneuver combined with several radio calls. The Aerodynamics & Guidance CSCI was then demonstrated in a stand-alone fashion, traversing over simulated flat terrain and viewed in several perspectives. Following the AGM, the CDE was demonstrated. Because of the maturity and visual nature of this component, a fair amount of detail was provided—beginning with the procedures to build an individual gauge, attaching it to a control panel, animating it, placing it in a cockpit, building a vehicle structure around the cockpit, and finally, placing the completed helicopter in the world prior to the simulation. As the last demonstration of the MIDAS specification capabilities, the Symbolic Modelling CSCI was returned to. This time it was used to portray a further decomposed mission with the design-dependent operator activities fully described as a result of the symbolic equipment models for the alternative communication switch configurations.

Jack was then used to perform some basic reach sequences in the AH-64A cockpit, using the maximum ranges of the human model data. The fact that the operator could not reach the panel-mounted communications panel when restricted to moving from the shoulder only (simulating a locked inertial reel) was demonstrated. Additionally, the potentially dangerous glare shield occlusion of the tailwheel lock and master arming switch was shown for "taller" pilots by attaching Jack's camera to the mannequin's eye during an animation sequence.

The Training Assessment CSCI demonstration was then conducted for two send-radio message tasks with the alternative radio switch configurations. The input, output, and processing characteristics of this component was described, as the attendees were shown how a knowledge-based system operates.

The newly initiated applied vision models were then introduced through a briefing. An Amiga-based prototype of the New York Association for the Blind's volume field of view model was used to render the binocular retinal maps, facial occlusions, and physiological blind spots for a series of mouse-selected fixation points.

The demonstrations were then concluded with the dynamic analysis capabilities of MIDAS. The fully populated cockpit was placed in the gaming area, driven by the aerodynamics and guidance model, and viewed from several perspectives. A summary output screen from a "simulation run" was then shown on the Symbolic Modelling color monitor. This screen showed the loading and task timelines for the two potential designs and made use of

different colors to describe physical resource conflicts between the scenario's flying tasks and the radio selection actions. Approaches for hypermedia-like access to Boff & Lincoln's *Human Engineering Data Compendium* (once it comes out on CD-ROM) was then shown through a simulated key-word search. The intent was to describe how analysts would be able to get extremely valuable context-sensitive information for areas such as "performance under vibration" or "effects of the use of gloves" to make cockpit design and mission decisions.

4.3 Programmatic Information

4.3.1 Risks

The majority of the risks faced by A³I during Phase III were management-oriented and not technical in nature. They stemmed from unclear and conflicting development direction combined with the fluctuating staffing situation. A program with the ambitions and scope of this magnitude cannot tolerate a continuance of these problems.

One quasi-technical risk does exist and should be mentioned however. It centers on the level of detail appropriate for the MIDAS design and analysis objectives. The Program Office has made it clear that MIDAS is intended for the conceptual development phase of crewstation design because of the high "payoff" for properly incorporating human engineering principles during this period. However, most of the known human performance models and analysis methods requires as inputs task, equipment, and environmental data which is more appropriate for detailed design. This apparent conflict between the model/analysis needs and the intended use of MIDAS is still unresolved. Its resolution will have serious implications for the Program's success in developing a prototype workstation which meets the needs of its projected users.

4.3.2 Summary of Results

The program had a tremendous response to the traditional end-of-phase demonstrations. Begun in November 1988, these demonstrations were attended by approximately 170 people from NASA, the US Army, other DoD components, as well as several universities. We were then asked by the Aeroflightdynamics Director to extend invitations to industrial sources, particularly the major helicopter manufacturers. The detailed demonstrations which resulted were actually conducted more as joint working groups and continued periodically through April 1989. This activity precipitated a significant amount of effort which can best be described as technology transfer. Lockheed Missiles and Space Company used our CDE package and vehicle dynamics interface to demonstrate a proposed Autonomous Underwater Vehicle concept. Boeing Commercial Aircraft Company spent three days at our facilities, understanding the tools, walking through code, and taking both software and documentation back to Seattle to set up a MIDAS-like design workstation at their company. Finally, the Marine Advanced Amphibious Attack Vehicle (AAAV) program office became very interested in the MIDAS capabilities, and we completed some vehicle prototyping design work for their review. Similar activities with the Fiber-Optic Guided Missile (FOG-M) program office and Boeing Helicopter Company also may evolve.

The few criticisms which were levied essentially boiled down to three areas. First, a number of people expected to see more explicit human performance models, especially in the cognitive area. Functions such as decision making, planning, scheduling, etc were not emphasized this phase. Even where present, such models were often embedded within the

mission decomposition/simulation component complicating their observation. Additionally, a number of attendees indicated they wanted more "hard analysis." People wanted to know specifically how MIDAS could enable them to find the "best" design in terms of any number of measures—both quantitative and qualitative. They also wanted to get to a "bottom line" in terms of mission success, etc. While "bottom line" aspects have never been a particular focus of MIDAS, significant room does exist for us to improve the analysis capabilities included for design evaluation. Finally, a number of folks dug deeply enough to see that we haven't yet reached the level of integration among the CSCIs which is intimated. Distinct equipment models exist on both the graphic and symbolic sides. Task information needed by the Training Assessment CSCI is not contained in the task objects under the Symbolic Modelling CSCI. Jack was only demonstrated in a stand-alone mode. The lack of a properly functioning simulation capability at the start of the demonstrations certainly contributed to this criticism. However, the point is generally accurate. The degree of integration among the CSCIs is not where it should be at this point in the overall development—primarily because it is the most difficult area of all to manage.

A number of significant design decisions were made during the phase as part of the development process. First, the previous plans for a multi-rate simulation executive (called the Modeller in Phase II) were dropped. While theoretically possible, the effort involved with supporting this approach outweighed the advantages. The ability to isolate and track state changes which spawned conditional behavior throughout any number of simulation objects was complicated by the fact that the tick resolution of individual objects may have been inadequate to perceive the triggering conditions. Forcing the models to all run at a nominal base rate was viewed as an acceptable and more manageable alternative, since at this stage in the program optimization considerations are not paramount. Secondly, the formal use of desired states as mission goals which were separate from primitive operator activities and operators is viewed as an important formalism to adhere to in future effort. This approach will allow our existing framework to make use of a great deal of the formal planning and goal directed behavior research ongoing at universities. Finally, the flexibility and power to describe the temporal and logical relationships between tasks was improved in a number of ways. Thirteen different temporal relationships were supported among tasks versus only parallel and sequential during previous phases. Additionally, inheritance restrictions between the children and parent nodes were removed allowing for a more realistic decomposition.

4.3 Limitations

One salient limitation of Phase III involved the lack of a comprehensive simulation capability. Because Phase III's detailed vertical slice into the mission entailed a radical change within the simulation objects for the operator tasks, some of the methods, constructs, and constraints used to support a simulation during previous phases were incompatible. The coding effort required to correct these problems was not devoted in time to instantiate a simulation capability with the new equipment models and component functions. Consequently, a task analysis window containing "simulated" simulation history information was used to convey our intended concepts. The absence of a full simulation capability also decreased the emphasis placed on integrating various components, such as the Aerodynamics & Guidance and Symbolic Modelling CSCIs, during this phase. Limitations such as these are to be expected however, particularly when a rapid prototyping approach is used and one attempts to pull together lots of disparate code under time pressure.

The MultiGen modelling package used as the underpinning for the CDE also brought out additional limitations, both programmatic and technical. First, a number of people were interested in using the CDE CSCI for various applications, but decided against it because of the relatively steep licensing fees. Additionally, MultiGen is a modelling package—not a

true CAD package. While this makes it easy and quick to use, it also precludes the production or sharing of "manufacturing quality" designs, complete with precise dimensions, etc. Because MIDAS is focused on the conceptual phase of development, this aspect was initially not thought to be a problem. However, much of the using community appears to want to use their actual CAD environment as a "home" for components such as JACK or applied vision models. In lieu of that approach, the desired alternative was to use the CDE output as their CAD input. Yet, data structure differences between MultiGen/CDE and typical true CAD packages makes this goal unachievable.

Finally, dimension errors and scaling problems arose when trying to share cockpit geometry information between the CDE and Jack CSCIs. While cockpits created using the CDE can be "saved" in a format compatible with Jack, importing them involves a considerable amount of tedious scaling and manipulation until they are proportional to the anthropometric model size. These artifacts will make any reach, fit, and visibility conclusions derived from these models questionable. The solution involves paying more attention to detail when rendering the cockpit and making minor translation code changes.

4.4 Future Directions

The focus for new MIDAS capabilities will be heavily influenced by the responses to our Phase III demonstrations, program office direction, related research findings, the funding outlook, and the capabilities of the staff. A couple of specific areas are known as of this writing. First, a dynamic simulation must be restored to the core MIDAS capabilities and the level of integration among the components must be expanded. Widely varying management priorities and staffing situations have disrupted the attention paid to these central features during Phase III—at a fairly significant expense. Secondly, emphasis must be placed on developing the Symbolic Modelling CSCI to a level of maturity and completeness commensurate with the other key components. Its role as the central human model environment and simulation forcing function demands that it receives a priority for resources and direction. Finally, during the next phase, the specific analytical and evaluation oriented aspects of MIDAS must be brought to the forefront. It must be clear how the MIDAS workstation improves the present iterative, man-in-the-loop design process.

These broad objectives will be realized through an internal project analogous to the AH-64 Apache Longbow program. Similar to the requirements of this on-going effort, we will attempt to achieve a more integrated Apache cockpit by incorporating functions currently performed with dedicated equipment into Multi-Function Displays (MFD). Using source data from the Longbow Apache MFD, the current AH-64 task analysis, and our 3-D graphic representation of the Apache, we will attempt to place the MFD into one of the crewstations and use the MIDAS capabilities to guide the design of the new display, as well as determine potential ramifications of its use in a simulated portion of a typical mission segment. This "glass cockpit" design and analysis emphasis dictates new requirements for almost every A³I CSCI which will hopefully become clearer in the months ahead.

5. HISTORICAL INFORMATION

5.1 Phase I Development

5.1.1 Requirements and Design Approach

5.1.1.1 Summary Level

The initial phase of development of the prototype HF/CAE workstation found the Program in serious danger of extinction due to insufficient funding caused by regular and sizable cuts from guidance funding levels. It was believed that a visually-compelling, proof-of-concept demonstration was required to communicate the essence of the Program to individuals unfamiliar with the particulars of the science involved. Maximum visual utility was demanded of every expenditure and development.

A baseline simulation capability was required that demonstrated mission modelling, human performance metrics and helicopter-pilot interactions within a controllable, time-stepped environment that provided multiple graphic "views" into the underlying model(s). The simulation needed to be incrementally extensible, hence only a framework for more elaborate modelling was required for this phase, given the time constraints imposed. Several areas of development emphasis were identified:

5.1.1.2 Mission Modelling

The overall A³I Program model architecture calls for a mission model driving the closed loop pilot-vehicle system. The model would be developed with a dynamic, interactive task analysis framework for systematically describing tasks involved in certain classes of advanced helicopter operations. The framework will provide a feasibility demonstration of the methodology, including all critical mission and flight management functions within a pre-specified sample scenario.

Typically, scout-attack helicopter missions are largely opportunistic or discretionary in nature. Consequently, the mission model generated by the mission decomposition methodology must allow this component to be represented, either by providing conditional branching based on some pilot model parameter, or stochastic event triggering.

Bolt, Beranek and Newman (BBN) had already started development (under a NASA contract initiated prior to the PDR) of a "Mission Decomposition Methodology" that would provide essentially the entire simulation structure for Phase I. Refer to the respective software component description document for the Phase I Mission Editor for details.

In order to drive and manipulate the specific mission model generated by the Mission Decomposition Methodology, a simulation executive was required that allowed greater user control of the simulation than conventional executive programs. Future integration of a vast number and variety of models within this executive structure was projected as well, hence flexibility as well as functionality was required. Refer to the software component description document for the Phase I Modeller for details.

Communications software was required to link the Symbolics 3670 running the mission model with the Silicon Graphics IRIS 2500T displaying dynamic, 3-D graphic views driven by the mission. The link was Ethernet under TCP/IP protocol. Refer to the software component description document for Phase I Communications for details.

5.1.1.3 Graphics

3-D, color, dynamic mission representation displays were required to provide intuitive understanding of simulation progress. Further, these so-called "views" became extremely valuable as debugging aids for programmers developing software on the system. Refer to the software component description document for the Phase I Graphic Views for details.

In addition to view graphics, a state display editor was required to allow designers to select appropriate model variables for run-time observation, and determine how and where the values were to be displayed. This tool allowed designers to individually select which simulation variables were of interest for monitoring, and the nature of their display (i.e. dial, bar, graph, etc.). Refer to the software component description document for the Phase I Icon Editor for details.

5.1.1.4 Human Performance Modelling

The first phase needed to demonstrate the capability to model, structure and analyze the human component of complex and interactive pilot-helicopter systems by elucidating the effects of human performance limitations on mission effectiveness. The mission had to be responsive to changes in pilot performance, and conversely, the pilot's loadings should be reflected in task loadings imposed by execution of the prescribed mission.

It was decided that emphasis should be placed on developing some meaningful demonstration of training effects as provided by profiles of novice and experienced pilot representations. Refer to the software component description document for Phase I Training Implications for details.

5.1.1.5 Demonstration Scenario

The demonstration scenario consisted of the capability to perform multiple consecutive simulations. Variables included:

- 1) A novice and experienced pilot profile that may be menu-selected prior to a simulation run to illustrate training effects.
- 2) Convoy return of missile fire at any point in the mission subject to the discretion of the designer.

It was possible to have extensive run-time control over the running of models from menu items. It was also possible to examine data and information both after the run, and during a model freeze state. Menu selection of these capabilities required no programming experience to start, operate and evaluate the simulation.

5.1.2 Hardware Environment

Figure 5 below indicates the Phase I hardware configuration. These components are described in further detail in the subsections which follow.

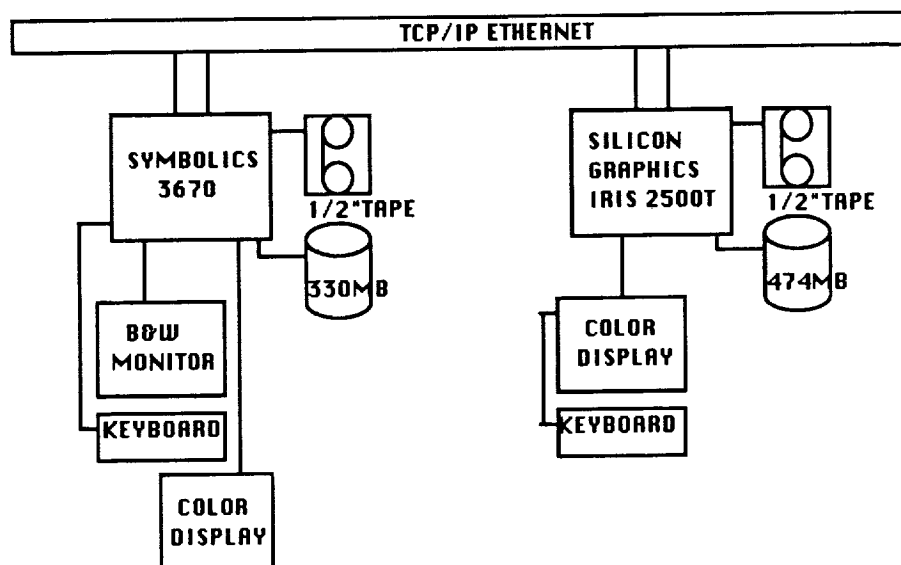


Figure 5. Phase I Hardware Configuration

5.1.2.1 Symbolics Lisp Machines

Model 3670-1433 Color Workstation consisting of:

- 8MB Main Memory
- Ethernet Controller and Transceiver
- 335 MB Fixed Disk
- Monochrome Console
- Keyboard & Mouse
- SYS36 20 MB System Software
- Documentation
- Model CG70-FB02 High Resolution, 24-bits/Pixel Color Frame Buffer
- Model CGOP-O1L 19" Color RGB Monitor
- Model OP36-FPA1 Floating Point Accelerator
- Model CGSW-PKG Software Package consisting of:
 - SCGR-DYNA Dynamic Animation System
 - SCGR-PAINT Paint System
 - SCGR-GEOM Geometry System
 - SCGR-RENDER Rendering System
- Symbolics # SLAN-FORT Fortran 77 Compiler
- Symbolics # STCP-1 TCP/IP Software

5.1.1.2 Silicon Graphics Computers

W-2500A Workstation consisting of:

- 1/4" Tape Drive
- HU-T04 Turbo Option W/4MB RAM
- H3-FPA Floating Point Accelerator
- H-DM4A 1024x1024x4 Display Memory
- H-ZC2 Z Clipping Assy,
- C-WTCP IP/TCP Software
- P-DBX Dial/Button Box
- R-UNIF Fortran Compiler

5.1.2.3 Other Processors

None.

5.1.2.4 Networking Hardware

TCL Incorporated Model 2010EC Ethernet Transceivers (tap-type)

5.1.2.5 Peripherals

Okidata Model 2410 Dot Matrix Printer

5.1.3 Software Environment

Generally, the graphic design and analysis tools were built using C, Unix, and the Replicore 3-D modelling package. Lisp-based tools and models were built using Symbolics Common Lisp under Genera 6.2, and the S-Packages were often used for Displays. Communication software was written to enable inter-machine message passing and simulation synchronization. The distribution of the Phase II models, tools, and displays is shown in Figure 3 below.

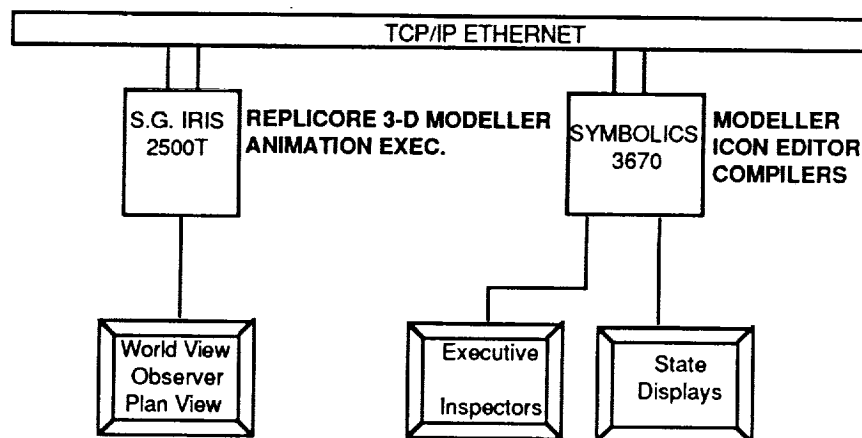


Figure 6. Phase I Software Modules

5.1.4 Programmatic Information

5.1.4.1 Risks

The majority of Phase I applications developments were under contract or subcontract. This condition poses some risk of failure caused by potential incompatibilities at the time of integration, competing priorities inherent within respective organizations, lack of control over developments and progress, and other problems that cannot be treated or corrected by the Program Office. While appropriate for this initial phase (due to constraints), each instance of reliance on organizations that the Program Office does not have direct control over should be carefully considered.

Since the Phase I architecture was minimal, there were few technical risks (outside of meeting deadlines) that might prohibit success. The major source of uncertainty involved networking the Symbolics and IRIS through TCP/IP Ethernet. Unfortunately, late delivery of graphics software forced a suboptimum (almost frenzied) approach to communications debugging, since in-house staff did not have the necessary familiarity with TCP/IP protocol details. Consultants were used to assist with debugging and optimizing the link.

5.1.4.2 Summary of Results

Preliminary demonstrations (of the mission model) were held in February 1986, followed by the first of several formal Phase I demonstrations starting in late June 1986. The demonstrations consisted of three computer displays (two in color):

- 1) Three-dimensional (3D), color, dynamic, mission representations composed of world, pilot and plan views of the simulated mission.
- 2) Mission model set-up, control and data display.
- 3) Color iconic "state displays" providing continuous display of mission model variable values.

The simulation executive (on the Symbolics 3670) controlling the mission model provided appropriate data via EtherNet TCP/IP to 3D graphic "views" resident on the IRIS 2500T to drive each dynamic simulation object. Mission model programming began in November of 1985, state display work began about the same time, while graphic display work started in late January, 1986. Over 50 man-months of combined programming effort was dedicated to this Phase, generating nearly 5000 lines of Fortran code (3000 in-house, 2000 contract), 1200 lines of C (in-house), and 7800 lines of Lisp (3800 in-house, 4000 contract) in this eight month period.

Numerous design and implementation compromises were made in the first phase of development due to the severe time constraints (start 11/1, complete by mid June), and minimal staffing available. Future phases must address more long-term strategic approaches.

5.2 Phase II Development

5.2.1 Requirements and Design Approach

5.2.1.1 Summary Level

Phase II was intended to devise more long-term approaches to the development of the workstation. Another primary purpose was to assemble a team of individuals (both in-house and outside) appropriate for this activity. This team would be composed of both researchers and implementers, since the Program's approach is to employ contemporary techniques to integrate the best models of human behavior/performance, vehicle/systems and environment available.

Phase I had succeeded in demonstrating that the concept of a prototype workstation for aiding early helicopter cockpit design with regard to human performance limitation was viable. The implications of attempting to develop such a system were also more clearly understood after Phase I. The purpose of Phase II was to:

- 1) Develop modelling and simulation tools

- 2) Develop graphics tools
- 3) Integrate 6 DOF helicopter dynamics
- 4) Develop a more representative mission
- 5) Design and implement a more modular architecture
- 6) Gain additional insight into modelling and the design process
- 7) Build an appropriate in-house implementation team
- 8) Establish working relationships with various centers-of-excellence

At the completion of Phase I it was evident that more long-term strategies to development would have to be adopted if the Program were to ultimately succeed. It was also recognized that a more active effort was necessary in the area of integration of research results if less mature fields such as human-computer interaction and predictive human modelling were to gain any acceptance.

New software development requirements in Phase II centered around modelling environment, pilot models, vehicle/systems models, world models, analysis and decision aiding, and user interfaces Work Breakdown Structure (WBS) elements. The specific applications chosen are described in the subsections that follow.

5.2.1.2 Modelling Environment

5.2.1.2.1 Mission Editor

Development of the Mission Editor continued in Phase II under subcontract with BBN with domain expertise and integration supplied by in-house staff. BBN designed a graphic editing interface utilizing the mouse and pop-up menu templates to relieve the user of Lisp code editing. A manual decision interface was also implemented to override the previous "selection by aspect" method of simulated pilot decision-making. The Mission/Task Editor is an application initially developed BBN that serves as the human performance modelling framework whereby more elaborate computational models of human behavior and performance may be installed or "activated" in the workstation contingent on the type and level of analysis required to answer a particular question. For example, the framework provides a default toplevel directed acyclic graph form of human task modelling that can be used to evaluate task sequencing and resultant human resource loadings (visual, auditory, cognitive and psychomotor) based on empirical data. However, it is possible to integrate detailed predictive computational models of human performance such as dynamic anthropometric models that are able to compute rates, durations, reach, comfort factors and other parameters. These detailed models can be utilized as an alternative to the more subjective toplevel empirical models that the system provides by default. The framework also provides interfaces to simulation models of the vehicle/systems and environment at various levels of abstraction. Most importantly, the framework provides contingent task behavior subject to vehicle and environmental state variables. Refer to Appendix 1 of the Phase II System Architecture Description Document (SADD) for details.

5.2.1.2.2 Modeller

The Modeller serves as the Phase II simulation executive. It provides all modes of interaction with the simulation, including build/edit, test/verify, experiment frame, run, analysis and document/report. The ultimate goal of the Modeller is to provide the complete environment for construction, integration, testing, simulation analysis and reporting of results. The most developed mode of the Modeller is the run mode, whereby the user controls the execution of the simulation models. Model selection, data specification and run-time display (state-displays and views) configuraton is provided through the experiment frame mode of the Modeller. Refer to Appendix 2 of the Phase II SADD for details.

5.2.1.2.3 Visual Modeller

The Visual Modeller is a prototype visual programming language for dynamic systems modelling developed under subcontract by Expert-Ease Systems Inc, of Belmont, CA. It allows the user to select components such as dividers, summers, integrators, sources and sinks from an extensible library of components and assemble them on a graphic "worksheet" to form working models. Connections between component input and output ports are made graphically, as is specification of initial conditions and parameters. Models are subsequently run as interpreted code (versus compiled) subject to boundary conditions (start time, end time, step size) supplied by the user. The application was initially developed as a stand-alone tool for evaluation, hence it has yet to be integrated with other MIDAS workstation elements. It is anticipated that some form of visual programming language will be developed for the Modeller build/edit mode that utilizes the same type of interface as the Visual Modeller. This tool would become the primary means of developing and integrating vehicle/systems and world models for the simulation. Refer to Appendix 3 of the Phase II SADD for details.

5.2.1.2.4 State Display Editor

The State Display Editor is an enhanced version of the Phase I Icon Editor, which is based on the Graphics Editor from Steamer, an application developed by BBN for the Navy Personnel Research and Development Center (NPRDC) in San Diego, CA. The State Display Editor is used to conveniently build displays composed of graphs, bars, dials, sliders, text, etc. for indicating the state of some selected dynamic variable during a simulation. The user interface has been substantially reworked to mimic the Apple Macintosh desktop metaphor. It also takes advantage of improvements to the Symbolics operating system (Genera 7.1) such as infinitely scrollable windows in both vertical and horizontal dimensions. Other features added to the original Icon Editor include "Macintosh-like" text handling, keystroke commands (to supplement pull-down menus) and numerous bug fixes that had existed from the original Steamer code. No programming is required to use this tool. Refer to Appendix 4 of the Phase II SADD for details.

5.2.1.3 Pilot Models

5.2.1.3.1 Anthropometric Model

The A³I Anthropometric Model was developed under a NASA/Ames grant to Dr. Norman Badler at the University of Pennsylvania's Department of Computer and Information Science. Based on Dr Badler's previous work, the POSIT/HIRES model provides human body dimensions, reach and position based on CAR (Crewstation Assessment of Reach, Boeing Corp.) database link sizes, driven by task/goal specifications (HIRES). Unlike other systems that statistically utilize anthropometric databases, POSIT allows the user to specify each link independently and establish motion constraints for any link or joint in the model. Since the anthropometric model is an ongoing research effort at the University of Pennsylvania, a working version of POSIT was not received by the Program until 1 month prior to the end of Phase II. Consequently, the capabilities of the system were demonstrated off-line as a stand-alone system, although the graphic databases generated by other MIDAS workstation tools were immediately made compatible with POSIT/HIRES, as well as the Mission/Task Editor's output. Future Phases of the Program will see integration of the next generation of POSIT/HIRES (JACK/GOALTENDER), which is expected to include dynamics and field-of-view modelling capabilities. Refer to Appendix 5 of the Phase II SADD for details.

5.2.1.3.2 Loading Model

The loading model currently used to measure human performance is based on data accumulated at the Army Research Institute at Ft. Rucker, AL, that was subsequently committed to computer hardware within a modelling framework developed by BBN. The model generally states that task performance requires human resources from visual, auditory, cognitive and psychomotor (VACP) dimensions. This approach is loosely based on Chris Wickens Multiple Resource Theory and subsequent model implementation by Aldrich & McCracken at Anacapa Sciences Incorporated. The data from ARI is based on a survey of active helicopter pilots who were asked to estimate the VACP loadings for various tasks on a scale of 0-7, given very specific guidelines and criteria for each level in that range. The advantages of this approach are visibility, simplicity, and intuitive appeal. The disadvantages are that VACP values obtained from the survey are context dependent, thus to model variations in loading as a function of vehicle/system or world state, as well as pilot variables (training level, stress, fatigue, workload) there is no empirical data available to support alteration of baseline VACP values as a function of these variables.

5.2.1.4 Vehicle/Systems Models

5.2.1.4.1 Dynamics and Guidance Models

The type of helicopter dynamics and guidance models used in Phase I were discrete point-mass. These models did not provide the necessary data in translational and rotational degrees of freedom to analyze vehicle orientation (pitch, roll, yaw) as a function of control movements. The dynamics model used in Phase II had been used at Ames previously in motion simulation studies on VAX VMS hosts, thus the majority of the effort required to use this model was a port to the Symbolics computer. This, however, was not a simple task due to compiler differences and the Program's need to "package" models in a modular fashion that simplifies integration and interaction with other code, most notably Lisp. The guidance routines were also based on existing code used in motion simulator facilities, although extensive enhancements were required and performed by Anil Phatak and Huan Tran of Analytical Mechanics Associates (AMA), Inc. These included outer-loop speed, horizontal and vertical path control feedback to provide path following that was more representative of real pilot strategies. An important lesson learned in Phase II was that it is not advisable to attempt to integrate models that are still undergoing development into a system that itself is evolving, particularly when there are language compatibility complications (e.g. Fortran and Lisp). Although this was necessary in Phase II due to time constraints, future Phases should avoid this through better planning.

5.2.1.4.2 Cockpit Display Editor

The Cockpit Display Editor (CDE) is a new application developed for the construction and editing of cockpit displays and controls. Its interface is based on the Multigen visual database editing program developed by Software Systems of San Jose, CA. Software Systems and Sterling Software entered a joint development agreement which allowed Sterling source code rights to the Multigen software in exchange for developing enhancements to Multigen (described in the Graphic Views SCDD). The CDE is 3-D, color, dynamic, and has a user interface modelled after the Apple Macintosh. Displays can be conveniently linked to model parameters, or animated from stored datafiles. The CDE contains the database of all positional and geometric attributes of displays and controls in the simulated cockpit. This database will eventually be utilized by human behavior and performance models such as signal detection and visibility to analyze optimal instruments/control placement.

5.2.1.5 World Models

5.2.1.5.1 World Models

World models have not changed significantly over Phase I, with the exception of replacing flat terrain and gaussian hills with Defense Mapping Agency (DMA) terrain data. Architecturally, considerable effort was devoted to separating the simulation of world objects from pilot and vehicle/systems simulation models in anticipation of migration to distributed or parallel processing hardware.

5.2.1.5.2 Views

Geometric representations of world objects have changed appreciably from Phase I with the addition of enhanced Multigen software. As mentioned in world models, flat terrain and gaussian hills have been replaced with (DMA) terrain data, and objects/vehicles have real-world dimensions since they were obtained from a simulator out-the-window visual system database (Singer-Link DIG). The DMA data can be read directly off a tape and transformed into a 3-D, colored image by the graphic modelling system. The simulation views have been enhanced as well, since Multigen features provide interactive, Macintosh-like editing of objects (it is an object-oriented editor) and subsequent viewing with 3 translational and rotational degrees of freedom. Model-driven or stand-alone animation of objects is also provided as a feature that was added to the basic Multigen software.

5.2.1.6 Analysis and Decision Aiding

5.2.1.6.1 Training Requirements Prediction

The impact of training was demonstrated in Phase I by providing 2 types of pilot models (skilled and novice) and comparing the performance results in a simulated mission for each case. The two models were based on the assumption that on the average, a less skilled pilot requires longer and perceives a higher (VACP) load in task performance than a skilled pilot. Phase II endeavored to demonstrate that it would be possible to perform an analysis of a simulated mission's tasks and extrapolate the training resource requirements necessary to train an individual to perform such a mission. The approach drew heavily from Instructional System Development (ISD) methodologies. Hence the focus was upon post-simulation analysis and training requirements estimation, rather than attempting to show the effects of skill level variations on mission performance. Both are considered important for the MIDAS workstation.

5.2.1.7 Demonstration Scenario

The Phase II Scenario involved walking demonstration attendees through a simulated mission and cockpit build, anthropometry analysis, and task loading/timeline inspection. Emphasis was placed on "running" as many components together in an integrated fashion for a slight derivative of the Ambush Scenario first started during Phase I.

5.2.2 Hardware Environment

Figure 7 below indicates the Phase II hardware configuration. These components are described in further detail in the subsections which follow.

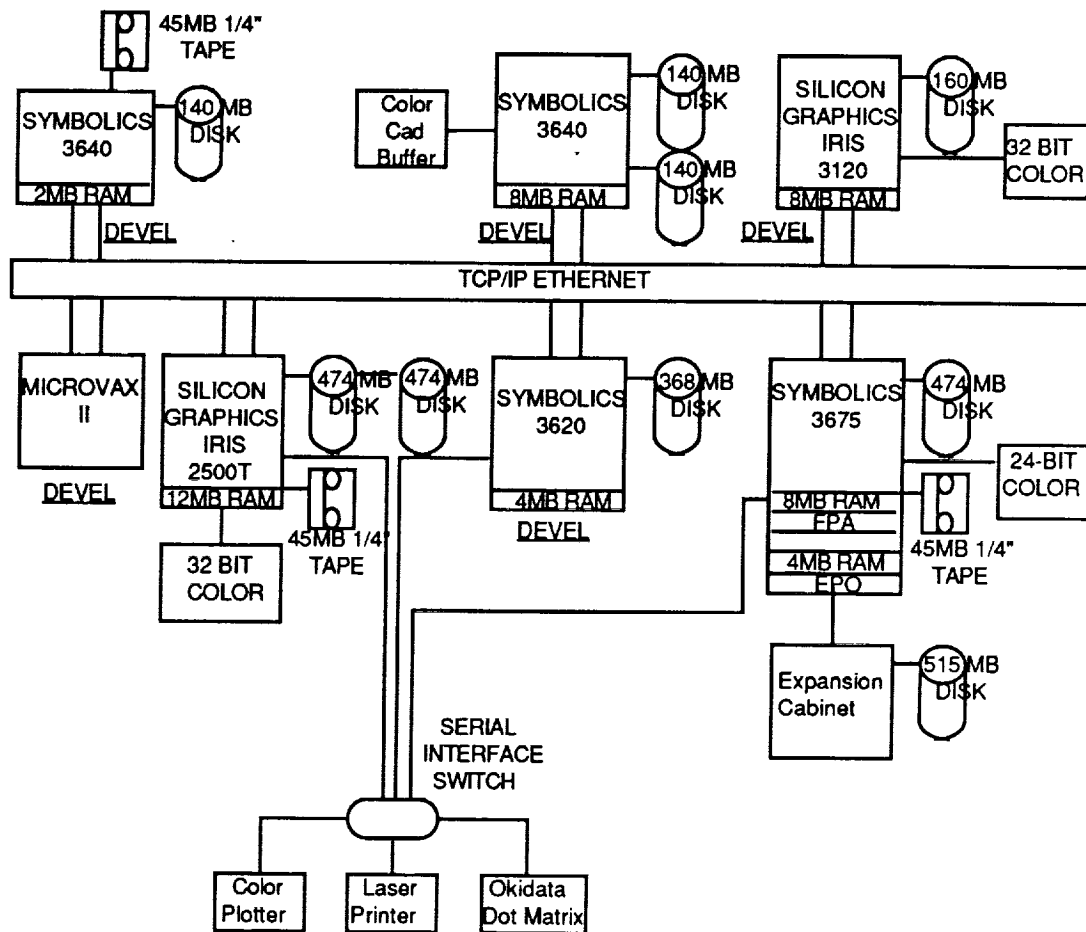


Figure 7. Phase II Hardware Configuration

5.2.2.1 Symbolics Lisp Machines

Model 3675 Color Workstation consisting of:

- Monochrome Console
- Keyboard & Mouse
- 45 MB 1/2" Cartridge Tape Drive
- Ethernet Controller and Transceiver
- 12 MB RAM
- Enhance Performance Option
- 474 MB Fujitsu Eagle Disk
- 515 MB CDC Disk
- Model CG70-FB02 High Resolution, 24-bits/Pixel Color Frame Buffer
- Tektronix 19" Color RGB Monitor
- Model OP36-FPA1 Floating Point Accelerator
- Symbolics # SLAN-FORT Fortran 77 Compiler
- Symbolics # STCP-1 TCP/IP Software

Model 3640 Color Workstation consisting of:

- Monochrome Console

Keyboard & Mouse
Ethernet Controller and Transceiver
6 MB RAM
2-140 MB Disks
CAD Buffer
Tektronix 19" Color RGB Monitor
Symbolics # SLAN-FORT Fortran 77 Compiler
Symbolics # STCP-1 TCP/IP Software

Model 3620 Monochrome Workstation consisting of:

Monochrome Console
Keyboard & Mouse
Ethernet Controller and Transceiver
4MB RAM
368 MB Disk
Symbolics # SLAN-FORT Fortran 77 Compiler
Symbolics # STCP-1 TCP/IP Software

5.2.2.2 Silicon Graphics Computers

W-2500A Workstation consisting of:

45 MB 1/2" Cartridge Tape Drive
Ethernet Controller and Transceiver
HU-T04 Turbo Option W/4MB RAM
H3-FPA Floating Point Accelerator
H-DM4A 1024x1024x4 Display Memory
H-ZC2 Z Clipping Assy
C-WTCP IP/TCP Software
P-DBX Dial/Button Box
R-UNIF Fortran Compiler

W-3120 Workstation consisting of:

Ethernet Controller and Transceiver
H3-FPA Floating Point Accelerator
H-DM4A 1024x1024x4 Display Memory
H-ZC2 Z Clipping Assy
C-WTCP IP/TCP Software
P-DBX Dial/Button Box

5.2.2.3 Other Processors

Digital Equipment Corp. MicroVax II

5.2.2.4 Networking Hardware

TCL Incorporated Model 2010EC Ethernet Transceivers (tap-type)

5.2.2.5 Peripherals

Okidata Model 2410 Dot Matrix Printer
Apple LaserWriter Plus Laser Printer
GraphOn Graphics Terminal

5.2.3 Software Environment

Generally, the graphic design and analysis tools were built using C, Unix BSD 4.3, and the IRIS Window Manager/Graphics Library II. The Multigen modelling package was used as the underpinnings for the CDE and Views components. Most Lisp tools and models were built using Symbolics Common Lisp under Genera 6.2, with the Flavors extension, and the S-Packages were often used for Displays. Communication software was written to enable inter-machine message passing and simulation synchronization. The distribution of the Phase II models, tools, and displays is shown in Figure 8 below.

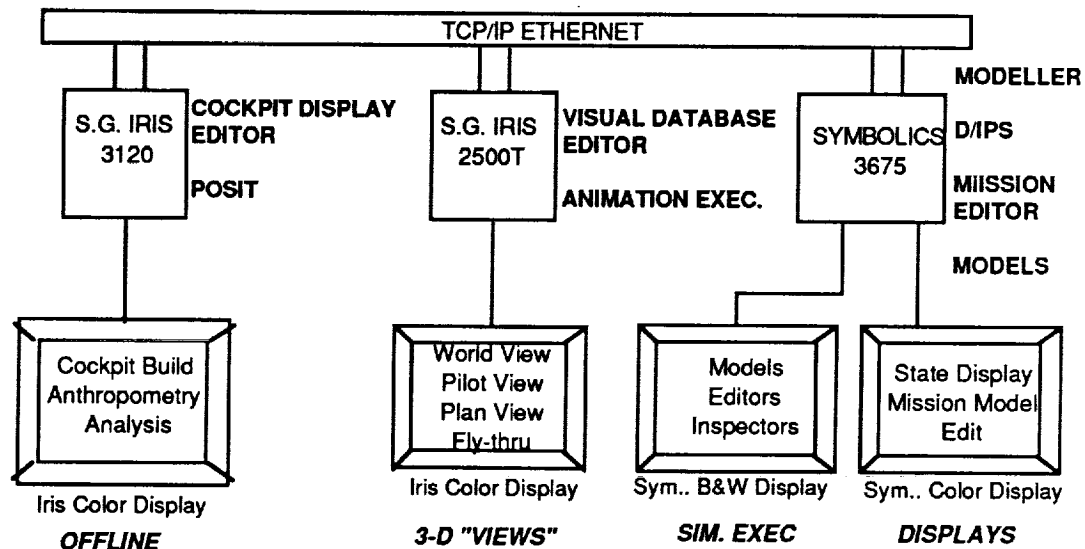


Figure 8. Phase II Software Components & Displays

5.2.4 Programmatic Information

5.2.4.1 Risks

By the end of Phase II there were 6 subcontracts and 1 NASA grant under the management of in-house implementation staff. This represents a sizable risk that the contributions of these subcontractors be appropriate, cost-effective and carefully monitored. The limited funds of the Program could not tolerate any waste of manpower or funds.

The decision to attempt to integrate discrete-event Lisp-type and continuous Fortran-type models in Phase II, without the benefit of a specialist in simulation modelling, could have proven to be a disaster. Although successful, the process was inefficient and frustrating. Future attempts at such model integration must be made with the aid of an expert in simulation modelling.

Hardware technological risks were again minimized by the use of standard, proven computers and software. Communications was TCP/IP protocol, and special hardware had been purchased to monitor network data packets in the event of any problems.

The composition of the in-house implementation staff is currently engineers, programmers, mathematicians, computer scientists, physicists and a helicopter pilot. There are no cognitive psychologists or human factors specialists. This is a serious void for a team

developing a prototype workstation for human factors analysis. Nonetheless, this void has been temporarily filled by subcontracting for this expertise.

5.2.4.2 Summary of Results

The first set of demonstrations was conducted for NASA and local personnel on October 22, 1987, followed by numerous scheduled and non-scheduled demonstrations over the course of the next 6 weeks. The feedback was without exception positive, although it became clear, after several comments, that in the next phase of development it would be necessary to address some concrete design issues rather than remaining general and abstract with regard to what the MIDAS workstation could accomplish.

Late in Phase II the Program Office obtained a Chief Scientist to direct ongoing research activities that are slated for eventual integration in the workstation. This addition tremendously improves the Program's ability to evaluate and respond to new research results that may be relevant to the design of complex man-machine systems. Similarly, two consultant subcontractors involved with the Program have extensive experience in the field of human behavior and performance modelling.

As Phase II was aimed at more long-term, strategic approaches, there will be a substantial amount of code reuse as well as coding momentum carried into Phase III. This momentum is further facilitated by reasonable success in assembling a team of individuals with appropriate skills and interests for this Program. These interests include human behavior/performance modelling, training, graphical modelling languages, integrative frameworks, decision aiding and user interfaces.

6. APPENDICES

6.1 Glossary, Definitions, Abbreviations

A ³ I	Army-NASA Aircrew/Aircraft Integration
AGM	Aerodynamics/Guidance Model
AMA	Analytical Mechanics Associates, Inc.
ART	Automated Reasoning Tool
BBN	Bolt, Beranek and Newman Laboratories, Inc.
CAD	Computer-Aided Design
CBT	Computer-Based Training
CDE	Cockpit Design Editor
CFT	Cockpit Familiarization Trainer
CPS	Computer Program System
CPT	Cockpit Procedures Trainer
CSCI	Computer Software Configuration Item
DTED	Digital Terrain Elevation Data
DMA	Defense Mapping Agency
DOF	Degrees-of-Freedom
EES	Expert-EASE Systems, Inc.
HF/CAE	Human-Factors Computer-Aided Engineering
I/O	Input/Output
ISD	Instructional Systems Development
MFD	Multi-Function Display
MIDAS	Man-machine Integration Design & Analysis System
NFS	Network File Software
NRC CoHF	National Research Council Committee on Human Factors

OFT	Operational Flight Trainer
SCDD	Software Component Description Document
SGI	Silicon Graphics Inc.
USGS	United States Geological Survey
WST	Weapon System Trainer

6. ANNEXES

ANNEX A — *SYMBOLIC MODELLING CSCI*

ANNEX B — *VIEWS CSCI*

ANNEX C — *COCKPIT DESIGN EDITOR CSCI*

ANNEX D — *ANTHROPOMETRIC MODEL "JACK" CSCI*

ANNEX E — *AERODYNAMICS/GUIDANCE CSCI*

ANNEX F — *COMMUNICATIONS CSCI*

ANNEX G — *TRAINING ASSESSMENT CSCI*

Annex A

Army-NASA Aircrew/Aircraft Integration Program

A³I

**Software Detailed Design Document:
Phase III Symbolic Modelling Software**

prepared by

Jerry Murray

December 1988

Table of Contents

1.0 INTRODUCTION.....	A-1
1.1 Identification.....	A-1
1.2 Scope.....	A-1
1.3 Purpose.....	A-1
2.0 RELATED DOCUMENTS.....	A-2
2.1 Applicable Documents.....	A-2
2.2 Information Documents.....	A-2
3.0 REQUIREMENTS AND DESIGN APPROACH.....	A-3
3.1 Background.....	A-3
3.2 Requirements and Rationale.....	A-5
3.3 Hardware Environment.....	A-6
3.4 Software Environment.....	A-6
4.0 DETAILED DESIGN DESCRIPTION.....	A-7
4.1 Organization.....	A-7
4.2 Basic Concepts.....	A-8
4.2.1 Global Variables.....	A-8
4.2.2 Entities.....	A-10
4.2.2.1 Entities Overview.....	A-10
4.2.2.2 The Entity Flavor.....	A-12
4.2.2.3 Static Entities.....	A-12
4.2.2.4 Dynamic Entities.....	A-13
4.2.2.4.1 The Dynamic-Entity Flavor.....	A-13
4.2.2.4.2 Functional Entities.....	A-14
4.2.2.4.3 Active Objects.....	A-15
4.2.3 Utilities.....	A-16
4.3 Environmental Modelling.....	A-17
4.3.1 Environmental Modelling Overview.....	A-17
4.3.3 Terrain Modelling.....	A-18
4.3.3.1 Terrain Objects.....	A-18
4.3.3.2 Digital Elevation Model (DEM) Arrays.....	A-20
4.3.4 Feature Modelling.....	A-21
4.3.7 Other Environmental Objects.....	A-21
4.4 Equipment Modelling.....	A-22
4.4.1 Equipment Modelling Overview.....	A-22
4.4.2 Equipment Modelling Objectives.....	A-24
4.4.3 Component Modelling.....	A-24
4.4.3.1 Component Modelling Overview.....	A-24
4.4.3.2 Functional Components.....	A-25
4.4.3.2.1 Equipment Component Basic Flavor.....	A-25
4.4.3.2.2 Equipment Component Example.....	A-27
4.4.3.3 Physical Components.....	A-29
4.4.3.3.1 Physical Component Basic Flavor.....	A-29
4.4.3.3.2 Physical Component Example.....	A-30
4.4.3.4 Component-Functions.....	A-31
4.4.3.4.1 Component Functions Overview.....	A-31
4.4.3.4.2 Component-Function Flavor.....	A-33
4.4.3.4.3 Component Function Example.....	A-33
4.4.5 Equipment Systems.....	A-35
4.4.5.1 Equipment Systems Overview.....	A-35
4.4.5.3 Equipment System Example.....	A-36
4.4.5.2 Helicopter Modelling.....	A-38
4.4.5.2.1 Helicopter Modelling Overview.....	A-38
4.4.5.2.2 Guidance and Aerodynamic Requirements.....	A-39

Table of Contents

4.5	Mission Modelling.....	A-40
4.5.1	Mission Overview.....	A-40
4.5.2	The Mission Flavor.....	A-40
4.6	Pilot Modelling.....	A-45
4.6.1	Pilot Modelling Overview.....	A-45
4.6.2	Pilot Definition.....	A-46
4.6.2.1	The Pilot Basic Flavor.....	A-46
4.6.3	Activity Representation.....	A-46
4.6.3.1	Activity Flavors.....	A-46
4.6.3.1.1	Test-Activity Flavor.....	A-46
4.6.3.1.2	Phase I and II Activity Flavors.....	A-48
4.6.3.1.3	Sequential Activities.....	A-49
4.6.3.1.4	Parallel and Parallel-stop Activities.....	A-49
4.6.3.1.5	Rotation Activities.....	A-49
4.6.3.1.6	Fixed Duration Activities.....	A-49
4.6.3.1.7	Intermittent Activities.....	A-49
4.6.3.1.8	Choice and Manual Choice Activities.....	A-50
4.6.3.2	Complex Activities.....	A-50
4.6.3.3	Activity Interactions with JACK.....	A-50
4.6.3.4	Activity Interactions with Aero Modelling.....	A-50
4.6.3.5	VACP Modeling.....	A-51
4.7	Task Decomposition.....	A-51
4.7.1	Task Decomposition Overview.....	A-51
4.7.2	Task Decomposition Display.....	A-52
4.8	Mission Simulation.....	A-57
4.8.1	Overview.....	A-57
4.8.2	Simulation Requirements.....	A-58
5.0	NOTES.....	A-58
5.1	Miscellaneous.....	A-58
5.2	Limitations.....	A-58
5.3	Future Directions.....	A-58
5.3.1	Activity Scheduling.....	A-58
5.3.2	Decision Modelling.....	A-59
5.3.3	Aircraft Guidance.....	A-59
5.3.4	Function Allocation.....	A-59
5.3.5	Mission Modelling.....	A-59

Figures

Figure 4.1	Symbolic Modelling SCI Organization	7
Figure 4.2	Entity Objects.....	11
Figure 4.3	Environmental Objects.....	17
Figure 4.4	Equipment Modelling.....	23
Figure 4.5	Transmitter Selector Switch.....	27
Figure 4.6	Component Functions.....	32
Figure 4.7	Communications Control Panel.....	36
Figure 4.8	Tactical Flight Profile.....	39
Figure 4.9	Pilot Model.....	46
Figure 4.10	Task Decomposition Display.....	53
Figure 4.11	Mission Simulation.....	59

1.0 INTRODUCTION

1.1 Identification

This document describes the Symbolic Modelling Software Configuration Item (SMSCI), which forms a part of the A3I Computer Program System. Descriptions of the detailed processing requirements, structure, I/O, and control are provided for each lower level Symbolic Modelling Software Component (SMSC), unit, or function contained within the SMSCI.

1.2 Scope

The material in this document is directed toward three categories of readers:

- 1) those who wish to learn what the A3I SMSCI does,
- 2) those who wish to use the Symbolic Modelling software to investigate the interactions between an operator and a specific crew station design within the context of a given mission,
- 3) those who might want to modify and update the Symbolic Modelling software.

This document attempts to describe the methodologies used to represent an object-oriented simulation environment in which the operator's activities are sensitive to crew station equipment design. Although some discussions of the mission, environment, general simulation issues and pilot activity modelling are provided, they are provided as background, detailed discussions of these issues are beyond the scope of this document. The primary focus of this document is to present the Phase III SMSCI equipment functional and physical modelling methodology. This methodology provides a means of explicitly demonstrating the sensitivity of specific pilot subtasks to equipment design.

Knowledge of the Symbolics programming environment and object-oriented programming is assumed along with a familiarity with Army aviation operations and task analysis.

1.3 Purpose

The purpose of the Symbolic Modelling Software Configuration Item is to provide a means of modelling a given design of a crew station, a required mission, the intended crew, the environments in which the crew station would be utilized, and the casual relations existing between these models. The models and their associated casual relations interact to develop a task decomposition which can be used for static design analysis or as a forcing function for a simulation using specific design, mission, crew, and environmental characteristics. The simulation and the resulting task histories are represented in the Simulation and Task Analysis SMSCs in a flexible format amenable to a wide range of analysis.

This document presents the results of work accomplished during Phase III of the A3I Program. It is important to understand that the primary objective for symbolic modelling in Phase III was to demonstrate how an operator's activities could be modelled to be "sensitive to changes in crew station design".

Many of the functions and data structures used were selected on the basis of how they contributed to achieving this objective. As is typical in any rapid prototyping environment, many of these functions and structures will not be appropriate in future phases. However, attempts have been made to present what is available in a manner which will support development in future phases.

A majority of the code developed in Phases I and II was not required to meet the current objectives and not incorporated into Phase III. A significant portion of this code, however, does address issues which remain major concerns of the A3I program and may be useful in future phases. Although this document does not intend to supercede previous documentation concerning code developed prior to Phase IV, the document does provide extensive information concerning the relationship of code developed in previous phases and code developed during Phase III.

2.0 RELATED DOCUMENTS

2.1 Applicable Documents

Symbolics Genera 7.2 Documentation, Symbolics Publication Number 999079, Symbolics, Inc., Cambridge, Massachusetts, 1988.

Development of an Advanced Task Analysis Methodogy and Demonstration for Army-NASA Aircrew/Aircraft Integration, BBN Laboratories, NASA Contract No. NAS2-12035

Engineering Data Compendium, Human Perception and Performance, Kenneth R. Boff and Janet E. Lincoln, Harry G. Armstrong Aerospace Medical Research Laboratory, Wright-Patterson Air Force Base, Ohio, 1988

Operator's Manual for Army Ah-64A Helicopter, TM 55-1520-238-10, Headquarters, Department of the Army, 28 June 1984.

A Computer Analysis to Predict Crew Workload During LHX Scout-Attack Missions, Anacapa Sciences, Inc. October 1984

2.2 Information Documents

A Representation for Complex Physical Domains, Sanjaya Addanki and Ernest Davis, Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, Ca, August 1985

Maintaining Knowledge about Temporal Intervals, James F. Allen, Communications of the ACM, 26(11), 1983

Setting up Large-Scale Qualitative Models, Brian Falkenhainer and Kenneth D. Forbus, Proceedings of the National Conference on Artificial Intelligence, Saint Paul, Minnesota, 1988

A Comprehensive Task Analysis of the AH-64 Mission with Crew Workload Estimates and Preliminary Decision Rules for Developing an AH-64 Workload Prediction Model, Anacapa Sciences, Inc. October 1986

3.0 REQUIREMENTS AND DESIGN APPROACH

3.1 Background

Descriptions of complex man/machine/environment interactions can be structured along any of the multiple dimensions of interest. Review of prior approaches to the analysis of helicopter mission performance yielded some insight into potential pitfalls in such analyses. Prior characterizations of helicopter missions generally follow a conventional timeline generation paradigm (Applied Psychological Services, 1982a,b,c; Sikorsky Aircraft ARTI documents, 1984a,b,c,d), and, in the case of Aldrich, Craddock and McCracken (1984), provide Monte Carlo simulated mission fly-outs for compilation of workload metrics. Conventional timeline analyses have several characteristics that make them inappropriate for the MIDAS workstation implementation. These deficiencies can be summarized as follows:

- Hard-coded representation does not provide for flexible modification of the mission.
- Large data bases are accumulated at a fixed, and generally very fine level of resolution. Modification of the mission must be performed at a level of detail inappropriate to the designer's main concern.
- In conventional mission analysis, the analyst may establish procedures or make assumptions in the decomposition of the mission that are generally difficult to identify. This reduces the potential for an audit trail of analyst's decisions.
- No common database has been established, therefore, there is no accumulation of knowledge as a function of continued analysis, and little chance of independent verification of the results of the analysis.

The Symbolic Modelling Software Configuration Item is designed for analysis of the man-machine interactions in complex crew station during a typical mission. A major requirement for the SMSCI is to integrate and interface the environmental, aircraft systems, mission, and crew models with other SCIs to produce a driving function for mission simulation. In order to do this, it is necessary to model the processes of the environment, equipment systems, mission and crew and the relational complexity of these processes.

- The requirements for environmental modelling include representation of terrain elevation, features such as roads, and miscellaneous vehicles and systems.
- Aircraft systems models are required to represent the basic aircraft and component systems at adjustable levels of detail.
- The mission model is the representation of actual mission requirements and associated doctrine and procedures as normally presented in a operations briefing. It is not a representation of the execution of a mission but rather a model which helps drive a mission simulation and the standard by which the results of a simulation are evaluated.
- Crew models are required to simulate various levels of training, experience, and performance capabilities. These models must provide a mechanism for explicitly representing the activities a crew member would performed in attempting to meet given mission requirements. This mechanism must be cable of representing activities at a level of detail appropriate to the designer's interests. The prime focus for the SMSCI in Phase III centered on making the crew member's activities sensitive to system equipment design.

Development of a dynamic and interactive analysis methodology for investigating environment, design, mission and crew interactions in the context of a mission simulation requires that the designer have the freedom to modify any of the significant elements being analyzed at varied levels of detail. In response, the simulation must provide for a reorganization and reordering of the tasks executed by the pilot, and a recalculation of performance and workload metrics as a function of the designer-imposed changes. Standard decomposition procedures tend to have a "Bottom-up" structure. The fundamental unit in these decompositions is a task which the aircrew must perform. Aggregation of these tasks make up the mission structure. The Symbolic Modelling SCI takes a different approach. In order to capture the characteristics that pilots bring to task performance, and to provide the flexibility described above, the SMSCI employs two interacting perspectives to guide the decomposition of the crew member's activities.

- The first perspective views the mission from the crew's goal structure in mission performance with a "Top-down" perspective. This goal structure explicitly represents the goal-subgoal relations and is sensitive to mission modification.
- The second "Bottom-up" perspective is that of a model based representation for the fundamental task units which are derived from the system design style based on functional and physical attributes. These fundamental task units define the actions a crew member may perform in attempting to satisfy a goal or subgoal.

These two organizing perspectives were designed to be mutually supportive. They also interact in a number of interesting ways. The methodology establishes two types of structures to represent the tasks to be performed in a mission: as goals/subgoal activities of the aircrew or as model-based task actions involved in each activity.

The SMSCI uses these structures to model a crew member's activities at an appropriate level of detail. The goal/subgoal activity structures provide a means of representing a crew member's ability to adapt his behavior to satisfy varying sets of goals. For some design problems, such as function allocation, precise details concerning a activity actions may not be needed and the goal/subgoal activity structures may be sufficient. For other design problems, specific details will be needed concerning task action attributes. In these situations, it is necessary to map each subgoal activity to task actions structures which represent a decomposition of the subgoal activity. These actions structures have explicit representations of casual models which provide task performance time, performance load, and other parameters which are dependent upon the functional/physical characteristics of the given man/machine system. This approach allows a designer to examine a mission from multiple perspectives while also providing a flexible system for representing environment, equipment, mission and crew models. Concepts developed in this phase included a method of relating operator activities to the appropriate equipment and a method of modelling the equipment so that the functional characterization does not constrain the physical implementations of a given function. This last concept allows a flexible environment for investigating alternative designs for a required function.

The SMSCI provides for the integration of environment, equipment, mission, and crew models as the forcing function for a tick-based mission simulation. The simulation has the form of a tree consisting of objects. Each object in it has a parent (or is the top-level object) and each object has component objects. At each beat of a driver clock, a tick message is sent to the top-level object (in this case, the A3I simulator). It performs whatever procedures it has been programmed to carry out during a tick, then passes the tick to each of its component objects. They carry out their tick procedures, then pass the tick downward, and so on. This approach was taken to meet the design goal of modularity. Anything that handles a tick message may be added to a list of component objects. This architecture allows for great flexibility and modularity in building simulations.

Some of the objects in an A3I simulation are active objects. An active object is an object is used to represent "intelligent" behavior and has a list of activities that it is carrying out. Examples of active objects in the A3I simulation are the pilot, helicopters and convoy vehicles (when their associated operators are not explicitly represented). Equipment systems may also be represented as an active object when it desired to model a specific behavior such as automated system functions. When an active object receives a tick message, it sends a tick to each of its activities. Methods were considered for implementing active objects at several levels of operational detail. This would allow the A3I workstation to support mission analysis at varied levels of detail. For example, currently, convoy activities are carried out only at the level of vehicular activity. In a larger-scale simulation one would want higher-level organizations of individuals -- platoons or divisions, for example -- to carry out activities that include creating and monitoring activities in their component objects. This would provide the potential for a simulation and analysis capability for theatre, mission, and sortie level operations using essentially the same software structures. Objects which are not active objects are able to respond to changes in state or the world by means of its tick procedures without the necessity of modelling activities as a component of the model.

3.2 Requirements and Rationale

The requirements for the Symbolic Modelling SCI are:

1. Symbolic representation of the environment, including terrain models and other vehicles and operators to the extent that they affect the primary operator's activities.
2. Symbolic representation of the operator's vehicle, crew station and associated subsystems.
3. Symbolic representation of a mission for a complex, rotary wing aircraft.
4. Symbolic representation of an operator performing tasks in a single-pilot, complex crew station in which the operator's activities are sensitive to the design of the crew station.
5. Integration of the symbolic operator model with anthropometric models.
6. Integration of the operator's vehicle with available aero models.
7. Integrate symbolic models with graphical models existing on other hardware by means of an independent communications program.
8. Produce a task decomposition and a simulation driving function which are dependent on the interactions of the mission, operator, vehicle and environmental models

The rationale the Symbolic Modelling SCI is built upon is based on the premise that the design of a crew station must be evaluated within the contexts it will be operating. It is necessary for the designer to be able to vary characteristics of the pilot, mission and environment in order to provide the necessary contexts needed to evaluate design alternatives. Although current features of the Symbolic Modelling SCI, and the A3I system in general, have been accepted as useful for evaluating various design alternatives, the evolving design has not been limited to purposes of evaluation but it intended to remain open to investigations of how the system may be useful in the conceptual design process.

3.3 Hardware Environment

The Symbolic Modelling SCI software runs on the Symbolics 3600 series under Genera 7.2. The associated simulation and task analysis CSCs required the Symbolics Color System hardware. The simulation displays used in Phase II ran on both HI-RES and CAD-buffer systems. A minimum of 8 megabytes of memory is recommended.

The SCI may also be run after recompilation on a MacIvory under Genera 7.3, however, the associated Task Analysis CSCI requires a Symbolics Color System which is currently not available on the MacIvory systems.

3.4 Software Environment

The Symbolic Modelling software is written in Symbolics Common Lisp (Genera 7.2) with extensive use of the Flavor System for object oriented programming and the Symbolics Dynamic Window and Presentation Substrate Systems.

The Symbolics TCP/IP System and the A3I Communication SCI is required for interfacing with Views, JACK and the Flight Dynamics/Guidance SCIs which run on the Silicon Graphics computers.

The associated simulation and task analysis modules required the Symbolics Color System software including the Image Management (IMAN) system. The color simulation displays used in Phase II ran on both HI-RES and CAD-buffer systems using software overlays effects that require refining the internal color map which translates pixel data to actual video color data. This may present conflicts with other windows using standard color map definitions.

4.0 DETAILED DESIGN DESCRIPTION

4.1 Organization

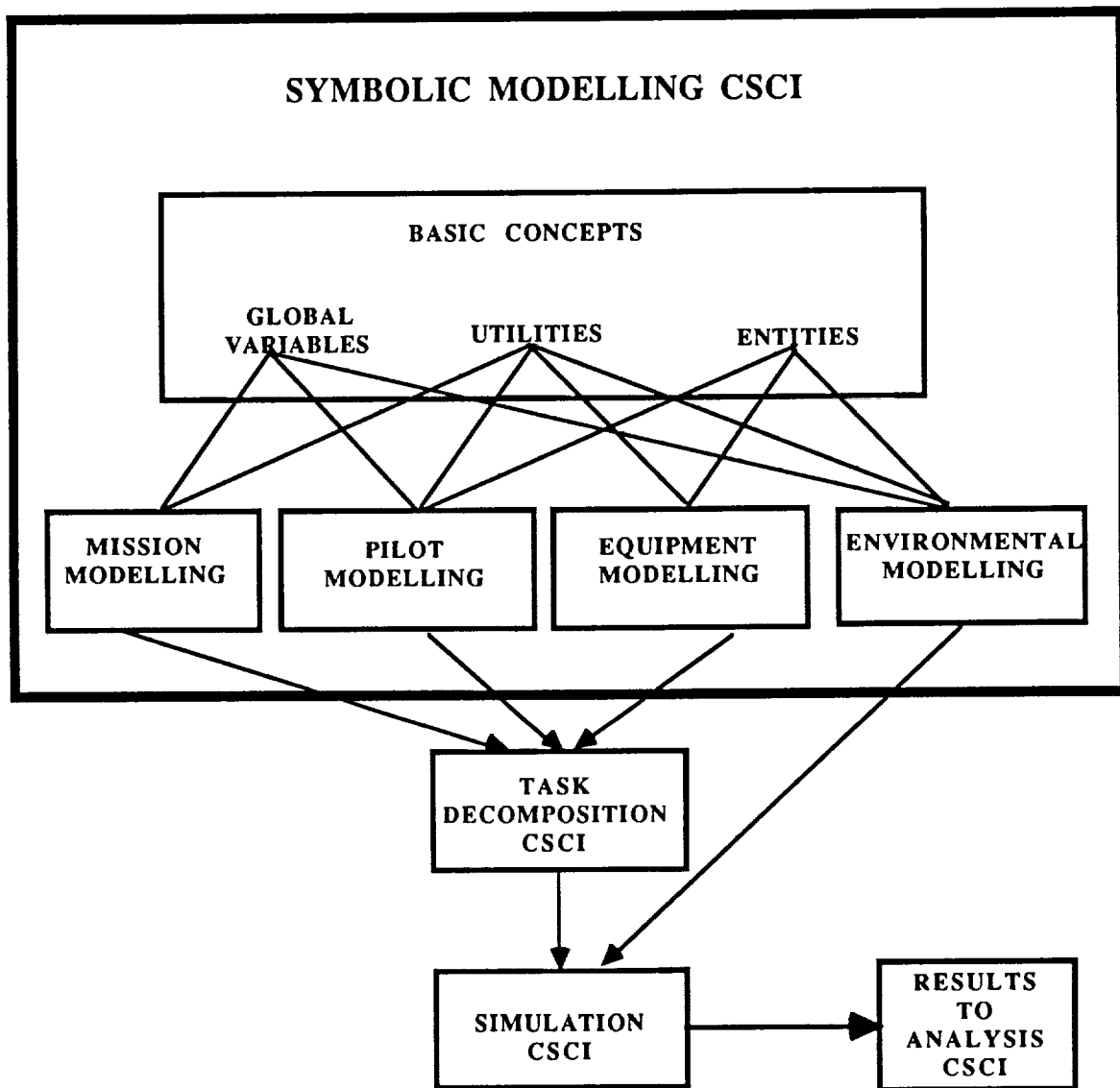


Figure 4.1 Symbolic Modelling CSCI Organization

4.2 Basic Concepts

4.2.1 Global Variables

Global variables are defined in the "CL-USER" package with an initial value of nil unless otherwise noted. These definitions are in the file "Barracuda:>p3>sys>basic>globals.lisp" which should be the first file loaded.

A3I-ENTITIES	List of all entity instances.
CURRENT-TIME	Time value used during a simulation. The simulation executive increments this value by sending a "TICK" message to all top level simulation objects.
DESIGN	Used to store information concerning different design alternatives. The functions for accessing information stored in the variable are presented below. Information may be stored with the associated LISP functions.

FUNCTION

RETURNS

(eval *DESIGN*)

Symbol representing different design alternatives.

NOTE: Phase III limited use of *DESIGN* to this method. See Section 4.6.5.3.2. The following mappings were used:

panel-radio-selection = a design with radio selection made with the transmitter selector switch on the instrument panel.

remote-radio-selection = a design with radio selection made by a remote switch on the cyclic.

(get *DESIGN* 'instance) An instance of a given design, such as, an instance of an AH-64 with associated subsystems.

(get *DESIGN* 'evaluation) A design evaluation summary or data structure used to store design evaluation information.

(get *DESIGN* 'all) A list of all known designs.

MISSION

A Mission Flavor object. See Mission Modelling in Section 4.5.

SEE-MOUSE-BLIPS	Used for mouse related activities in the task decomposition frame. Changes in future phases should enable this variable to be discarded. See future directions in Section 4.6.5.3.2.
TASK-DECOMP-FRAME	The constraint frame used for displaying task decomposition. See Section 4.6.5.3.2.
TD-TOP-MENU	Used by the task decomposition constraint frame for selecting menus. Changes in Phase 4 should enable this variable to be discarded. See future directions in Section 4.6.5.3.2.
TEST-ACTIVITIES	List of all test-activity instances. See Section 4.6.3.

The following variables were used in Phase III for demonstration purposes only and will not be used in future phases.

A1	List of activities representing the results of a simulation using a design with radio selection made the transmitter selector switch on the instrument panel.
B1	List of activities representing the results of a simulation using a design with radio selection made the remote selector switch on the cyclic.

The following variables were used in the Phase II and are specific to an area in which the point of origin was arbitrarily defined as a point 10 miles south and 10 miles west of the simulation gaming area. See the VIEWS CSCI Document for more details.

LOWER-LEFT-X-IN-FEET	64741
LOWER-LEFT-Y-IN-FEET	65621
X-CONVERSION-VALUE	10.33
Y-CONVERSION-VALUE	12.67

4.2.2 Entities

4.2.2.1 Entities Overview

Entities are static, functional, active or composite objects which exist within the context of a simulation. Static entities possess attributes which are not affected by the simulation and are used to represent objects such as the terrain and roads. The attributes of a static entities may vary from simulation to simulation as desired by the designer or by the problem definition but, once a simulation is initialized, these attributes would not be affected by interaction with other entities and a static entity may be viewed as an object whose state does not change during simulation. If it is necessary to change attributes of an entity during simulation in order to represent dynamic properties or behavior, the entity should be represented as a dynamic entity which are further defined as functional or active.

Functional entities represent objects which change state during simulation according to clearly defined functional models that are able to indicate state changes at the desired level of detail. Due to the complexity of the systems in a crew station, these functional models will in most cases need to be qualitative models. In many situations, it is necessary to explicitly represent causal relationships between models.

Active entities have a complex mechanism for changing state during simulation which both provides flexibility for defining the entity's behavior and a means of accessing details necessary for analyzing various aspects of the object's behavior.

Whether an object should be represented as a static, functional or active entity depends more on the objectives of a simulation than the specific nature of the entity. Static entities require significantly fewer resources during a simulation than dynamic entities and should be used whenever possible. The difference between functional and active entities is primarily the mechanism used to change the attributes which represent the state of the entity during simulation. Active entities provide flexibility and greater detail but also require significantly greater resources.

Composite entities are defined in context of their components which may be static, functional or active. Composite objects should be defined using functional or active entity flavors as components depending on the behavior desired.

The Symbolics Flavors system is used to represent entities within the symbolic modelling area of the A3I system. The entity flavors are used as base flavors to provide basic attributes and operations. The hierarchy of base items dependent upon the Entity flavor is shown below. This hierarchy is modified depending upon whether the environment, the design of the crew station, or the pilot is being modelled.

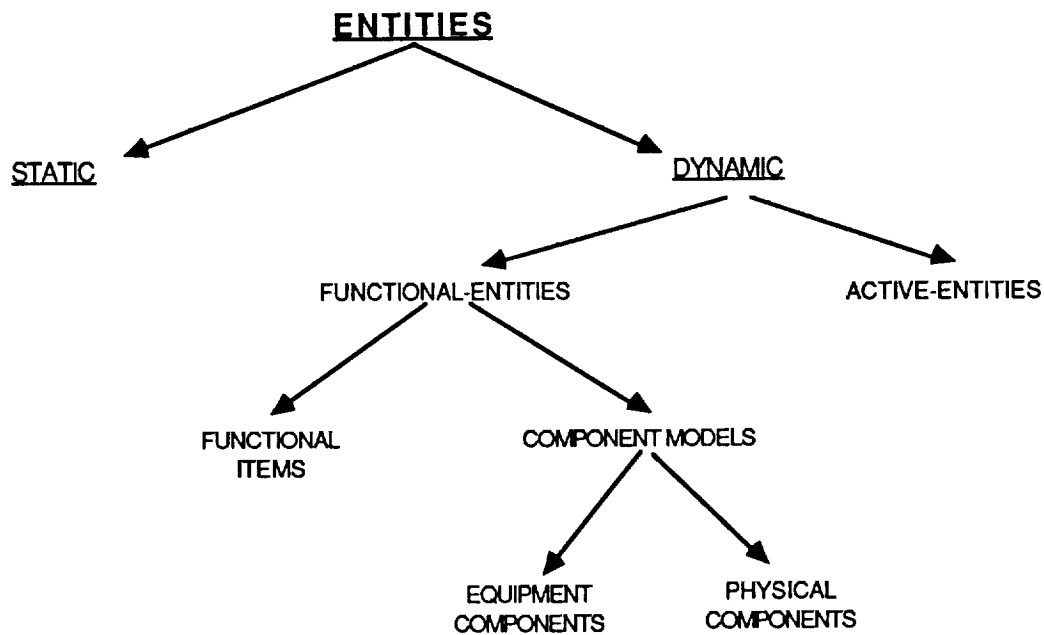


Figure 4.2 Entity Objects

4.2.2.2 The Entity Flavor

ENTITY flavor

Defines the base flavor for ENTITY objects. Entity objects represent objects which exist within the context of a simulation.

The following instance variables are used:

name----->name of most specialized flavor component.
 entity-type----->Static, functional or active (defaults to 'static').
 If the object is defined as a dynamic or active object,
 then the entity-type will be set by either a functional
 or active flavor component for the instance by
 means of the default-init-plist.
 location ----->this variable's structure is dependent on how we
 integrate with the CDE models.

The BASIC-ENTITY flavor is a base flavor and does not have any specialized flavor components.

(MAKE-INSTANCE BASIC-ENTITY :AFTER) method

Purpose: Provides means of accessing any entity through the global variable
 A3I-ENTITIES which is list of entity instances.
 Called by: (MAKE-INSTANCE DYNAMIC-ENTITY)
 Returns: N/A
 Side-affects: Adds each entity instance to list of entities in the global variable
 A3I-ENTITIES

4.2.2.3 Static Entities

Static objects are used for representing entities which contain information necessary for the simulation but which do not change state during the execution of a simulation. The terrain and roads are examples of static objects. Static objects are defined with flavor definitions which include BASIC-ENTITY as a flavor component.

4.2.2.4 Dynamic Entities

4.2.2.4.1 The Dynamic-Entity Flavor

DYNAMIC-ENTITY flavor

Defines the base flavor for DYNAMIC-ENTITY objects

The following instance variables are used:

current time ----->simulation time.

cycle-flag ----->not currently used. May be used to ensure all simulation tick procedures have completed.

The following instance variables are inherited from BASIC-ENTITY

name ----->name of most specialized flavor component.

entity-type----->Indicates entity type. Should not be set in this flavor since it will be set by either a functional or active flavor component for the instance by means of the default-init-plist.

location ----->this variable's structure is dependent on how we integrate with the CDE models.

This flavor has BASIC-ENTITY as a flavor component.

(TICK DYNAMIC-ENTITY :BEFORE) method

Purpose: Updates the objects simulation time value as the first action when the object receives a TICK message.

Args: None

Called by: (TICK ENTITY)

Returns: N/A

Side-affects: Increments current time instance variable

(TICK DYNAMIC-ENTITY) method

Purpose: usually not called because of method inheritance.

Args: None

Called by: Called as required.

Returns: N/A

Side-affects: Enables before and after methods to run.

(UPDATE-INSTANCE DYNAMIC-ENTITY) method

Purpose: Provided to enable before and after methods if a tick method is not provided by the flavor which use DYNAMIC-ENTITY as a component flavor.

Args: None

Called by: (MAKE-INSTANCE EQUIPMENT-COMPONENT :AFTER)

Returns: N/A

Side-affects: Enables before and after methods to run.

4.2.2.4.2 Functional Entities

Defines the flavor for FUNCTIONAL-ENTITY objects

The following instance variables are used:

- geometric-prop ----->this variable's structure is dependent on how we integrate with the CDE models. This variable is not being currently used.
- functional-objects ----> a list of object names. These objects have the potential to be functional during a simulation. The concept of a generic object needs to be implemented to provide access concerning subcomponent objects if they are not instantiated as functional-instances.
- functional-instances -->a list of objects which have the potential to be functional during a simulation.
- functional-flag ----->flag used to activate functional objects. If this variable is set to 'off or 'on-components-off at the time the object is instantiated, the objects specified in functional-objects will not be instantiated.

The following instance variables are inherited from DYNAMIC-ENTITY

- current time ----->simulation time.
- cycle-flag ----->not currently used. May be used to ensure all simulation tick procedures have completed.

The following IVS are inherited from BASIC-ENTITY through DYNAMIC-ENTITY.

- name ----->name of most specialized flavor component.
- entity-type----->Set to 'functional in the default-init-plist.
- location ----->this variable's structure is dependent on how we integrate with the CDE models.

(MAKE-INSTANCE FUNCTIONAL-ENTITY :AFTER) method

- Purpose: Creates instances of functional components when an object is instantiated unless the value of the IV functional-flag is 'off or 'on-components-off.
- Args: Ignored
- Called by: (MAKE-INSTANCE DYNAMIC-ENTITY)
- Returns: N/A
- Side-affects: Sets functional-instances IV to instances of functional components.

(TICK FUNCTIONAL-ENTITY :AFTER) method

- Purpose: Send a tick message to the functional component objects after the local tick methods have completed.
- Args: None
- Called by: Called as required.
- Returns: N/A
- Side-affects: Enables before and after methods to run.

4.2.2.4.3 Active Objects

Defines the flavor for ACTIVE-ENTITY objects

The following instance variables are used:

activities----->List of activities. In Phase II, all activities in this list were active and it was not possible to indicate known future activities. In future development, it will be possible that have this list to represent both active and future activities by adding an attribute to the activity objects representing their status. This will be important for planning functions.

activity-history----->List of activities. In Phase II, this represented all activities which were active or had been active. In the future this should also include other activities, such as required activities which were not performed.

The following instance variables are inherited from DYNAMIC-ENTITY

current time ----->simulation time.

cycle-flag ----->not currently used. May be used to ensure all simulation tick procedures have completed.

The following IVS are inherited from BASIC-ENTITY through DYNAMIC-ENTITY.

name ----->name of most specialized flavor component.

entity-type----->Set to 'active in the default-init-plist.

location ----->This variable's structure is dependent on how we integrate with the CDE models and is not currently being used.

(TICK ACTIVE-ENTITY :AFTER) method

Purpose: Controls running the activities in the activity list instance variable.

Args: None

Called by: Called as required.

Returns: N/A

Side-affects: Send a tick message to the activities objects after the local tick methods have completed.

4.2.3 Utilities

The following functions are defined in the file "Barracuda:>p3>sym>basic>utilities.lisp". The functions are general utilities which may be called as needed.

CONVERT-X-IN-FEET-TO-PIXELS function

Purpose: Finds pixel value for a value for X given in feet.
Args: An integer representing longitude (X) in feet.
Returns: Corresponding pixel value for longitude.
Notes: This function depends on corresponding values for *lower-left-x-in-feet* and *x-conversion-value* in order to return appropriate pixel values.

CONVERT-Y-IN-FEET-TO-PIXELS function

Purpose: Finds pixel value for a value for Y given in feet.
Args: An integer representing longitude (Y) in feet.
Returns: Corresponding pixel value for longitude.
Notes: This function depends on corresponding values for *lower-left-y-in-pixels* *lower-left-y-in-feet* and *y-conversion-value* in order to return appropriate pixel values.

CARTESIAN-DISTANCE function

Purpose: Finds 2d distance between points
Args: x1 y1 x2 y2
Returns: Integer representing distance between points.

INTERPOLATE function

Purpose: Perform linear interpolation to point 3 from points 1 and 2.
Args: x1 y1 x2 y2 x3
Returns: Value for y3.
Note if x1 = x2, y1 is the value returned.

CONVERT-TO-DEGREES function

Purpose: Find the angle in degrees for the given dx and dy.
Args: Dy and Dx
Returns: Angle in degrees.

4.3 Environmental Modelling

4.3.1 Environmental Modelling Overview

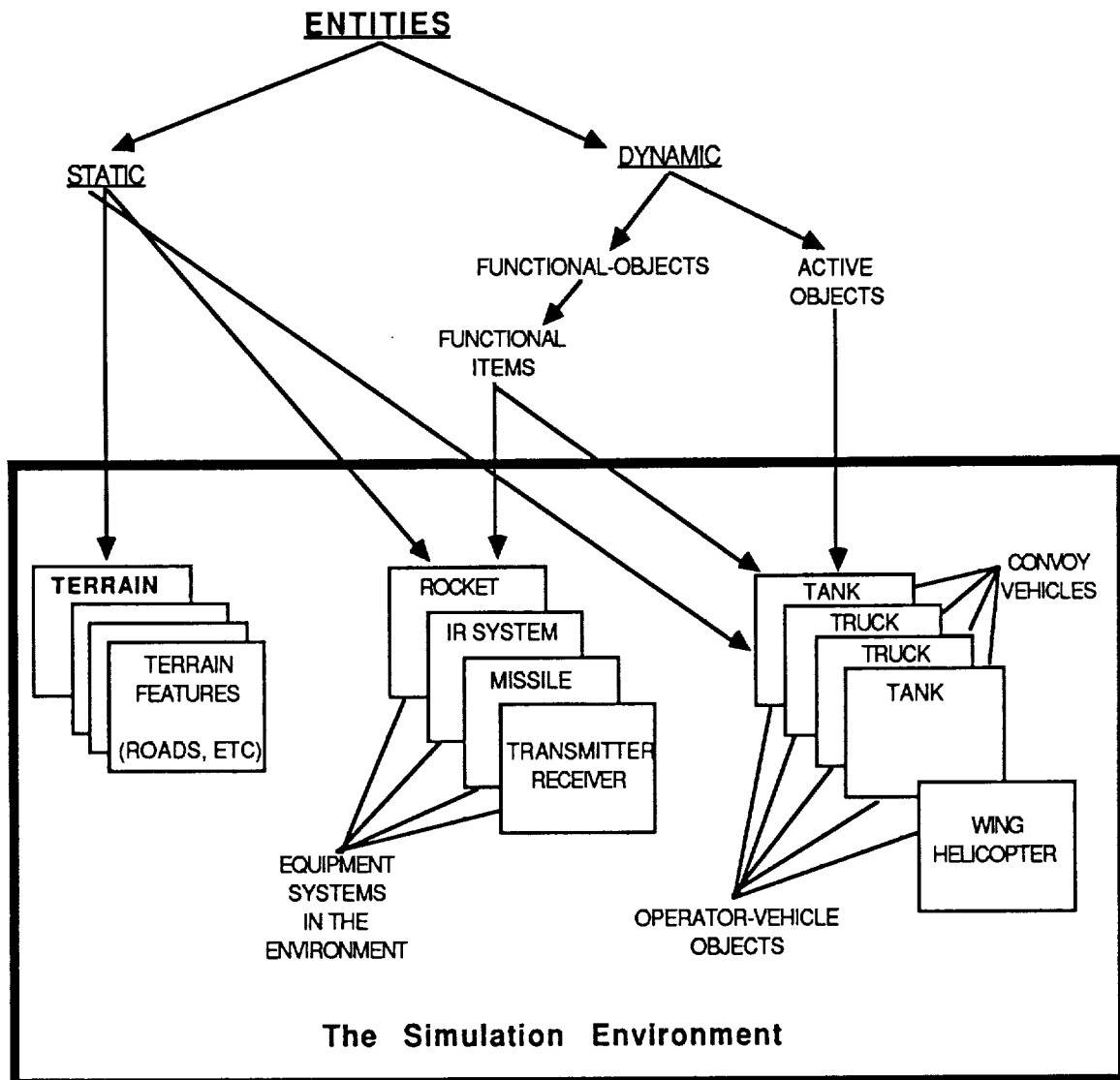


Figure 4.3 Environmental Objects

4.3.3 Terrain Modelling

4.3.3.1 Terrain Objects

Terrain modelling is currently performed on both the Symbolics machines and the Silicon Graphics IRIS machines. Initial terrain data is obtained from DMA files by a program running on an IRIS (See the VIEWS documentation) and supplied to the Symbolics as a data file used to initialize an array. Modelling is currently being done on both machines to reduce communication requirements.

TERRAIN flavor

Defines flavor for TERRAIN objects

The following instance variables are used:

- ground-los-angle---->Lowest permissible line-of-sight angle.
The method for determining ground-los-sightt uses this value. This value is currently static and is used to address the problem associated with looking at points over the horizon. The ground-los-sight method does correctly address downslope situations.
- los-increment----->Line of sight increment. Specifies distance between intermediate points which are checked when determining if line of sight exists between two points.
- dem----->Digital Elevation Model (DEM). Two dimensional array storing elevation values as an integer for given x and y.

(ELEVATION TERRAIN) method

- Purpose: Determine elevation at a given x and y.
- Args: Integers values for x and y world coordinates.
- Called by: Procedures in activities.
- Returns: Integer representing elevation.

(ELEVATION-FROM-PIXELS TERRAIN) method

- Purpose: Determine elevation at a given x and y.
- Args: Integers values for x and y screen coordinates.
- Called by: User interface procedures.
- Returns: Integer representing elevation.

(LINE-OF-SIGHT? TERRAIN) method

- Purpose: Determine if line-of-sight exists between two points
- Args: X, Y and Z world coordinates for two points.
- Called by: Procedures in activities
- Returns: T or nil

(GROUND-LOS-CHECK TERRAIN) method

Purpose: Determine if the horizon prevents los.
Args: X, Y and Z world coordinates for two points.
Called by: (LINE-OF-SIGHT? TERRAIN)
Returns: T or nil

(BLOCKING-TERRAIN? TERRAIN) method

Purpose: Determine if terrain blocks los.
Args: X, Y and Z world coordinates for two points.
Called by: (LINE-OF-SIGHT? TERRAIN)
Returns: T or nil

(PATH TERRAIN) method

Purpose: Find intermediate points between two points.
Args: X, Y and Z world coordinates for two points and step size between intermediate points.
Called by: Procedures in activities
Returns: A list of intermediate points in the following format:
 ((Xa Ya Za) (Xb Yb Zb) (Xn Yn Zn))

(NOE-PATH TERRAIN) method

Purpose: Find intermediate points between two points in which the z values of the intermediate points are elevation values determine by the point's x and y.
Args: X, Y and Z world coordinates for two points and step size between intermediate points.
Called by: Procedures in activities
Returns: A list of intermediate points in the following format: ((Xa Ya Za) (Xb Yb Zb) (Xn Yn Zn))

4.3.3.2 Digital Elevation Model (DEM) Arrays

LOAD-MAP-ARRAY-2 function

Purpose: Loads terrain array with values from DMA data
Args: Two arrays:
 1. The initial terrain array structure to be loaded
 2. An array providing values from DMA data.
Called by: Top level function
Returns: N/A
Side-affects: Load terrain array with initial values.
Note: Function includes a declaration for a sys:array-register to improve performance. Also, it is important to make sure the input array represents data in a compatible array structure. A typical error would be having an array which maps values for Y with the origin of Y at the lower left to a structure which represents the origin of Y at the upper left.

FILL2-MAP-ARRAY-2 method

Purpose: Interpolates elevation values from DMA supplied values to provide resolution needed for contour smoothing.
Args: A terrain array.
Called by: Top level function.
Returns: N/A
Side-affects: Fills values in terrain array.
Note: A poor kludge that does not fill the last column correctly.

FILL-LAST-COLUMN-MAP-ARRAY-2 function

Purpose: Fix brain damage code in fill2-map-array-2 and fill2-map-array-2.
Args: A terrain array
Returns: N/A
Side-affects: Fills last column in terrain array.

FIND-HIGH-LOW function

Purpose: Find high and low values in a array
Args: A 771 x 768 terrain array
Called by: Top level function
Returns: A list in the form (<high> <low>)
Side-affects: None

4.3.4 Feature Modelling

Features of the environment, objects such as roads, are with the ENTITY flavor as an component.

PATH flavor

Defines flavor for PATH objects

The following instance variables are used:

turn-points----->a list of points defining a route.

epsilon----->distance value in determining if
point has been reached.

The following instance variables are inherited from the ENTITY flavor.

name ----->name of most specialized flavor component.

entity-type----->Static, dynamic or active.

location ----->this variable's structure is dependent on
how we integrate with the CDE models.

(NEXT-POINT? PATH) method

Purpose: Where do I go on the path from my current position? We assume
that the last-point is given, and is one of the turn-points.

Args: last-point

Called-by: Mission activity procedures.

Returns: A turn point list.

Side-effects: None

4.3.7 Other Environmental Objects

Objects in the environment include the wing helicopter, convoy vehicles, missiles and similiar objects. Since Phase III investigated issues concerning actions inside the crew station, it was not necessary to model environmental objects and the code for environmental objects has not been modified in this phase. Significant changes will be necessary to adapt the code for future simulations. Phase II code for environmental objects is located in the directory named "B:>A3I>models>world.directory".

4.4 Equipment Modelling

4.4.1 Equipment Modelling Overview

The A3I system is designed for investigating human interaction with complex systems within the context of specific mission objectives and environmental conditions.

With this perspective, it should be noted that the Equipment Modelling unit limits its scope to the equipment currently under investigation and other equipment necessary for a mission simulation (i.e. other friendly helicopters, threat vehicles, etc.) are managed by the Environmental Modelling unit.

By design, equipment unit objects are modelled in a manner which permits easy interface with pilot/crew models which control state changes normally associated human interaction (i.e. turning a switch off). The equipment unit objects require external operator models for human controlled state changes. On the other hand, environmental unit objects may have an operator implicitly represented as part of the equipment model itself. This was demonstrated in previous phases with the convoy vehicles in which the vehicle objects had the ability to follow a route without explicitly representing the operator. Since Equipment Modelling unit objects require significantly greater computational resources, the ability for the designer to select varying levels of modelling effort provides the designer with a means of controlling how system resources are allocated. The decision of whether a specific system or component is modelled as an Equipment unit object or as an Environmental unit object is not based on the attributes of the equipment but rather by the objectives of the designer.

The usual method for modelling is to model one helicopter and all of its subsystems as Equipment unit objects and other helicopters and systems as Environmental unit objects. An example of a typical deviation from this pattern would be in a situation where the designer is concerned with the interaction of the flight and communication systems. In this situation, the designer may decide to model the helicopter's weapons systems as environmental objects if it is felt the systems are necessary for the mission simulation but not actually a factor in the designer's concerns of the interaction of the flight and communication systems.

Equipment items may be modelled as components or systems and, in some cases, represented concurrently as both. The context in which information concerning an item is required should determine if the information is represented in a component or system model. Components are always considered as part of an equipment system. A distinguishing characteristic of components is that from within the appropriate context a single functional/physical model provides sufficient information concerning the functional/physical characteristics of the item.

Details of components and systems are given in the following sections.

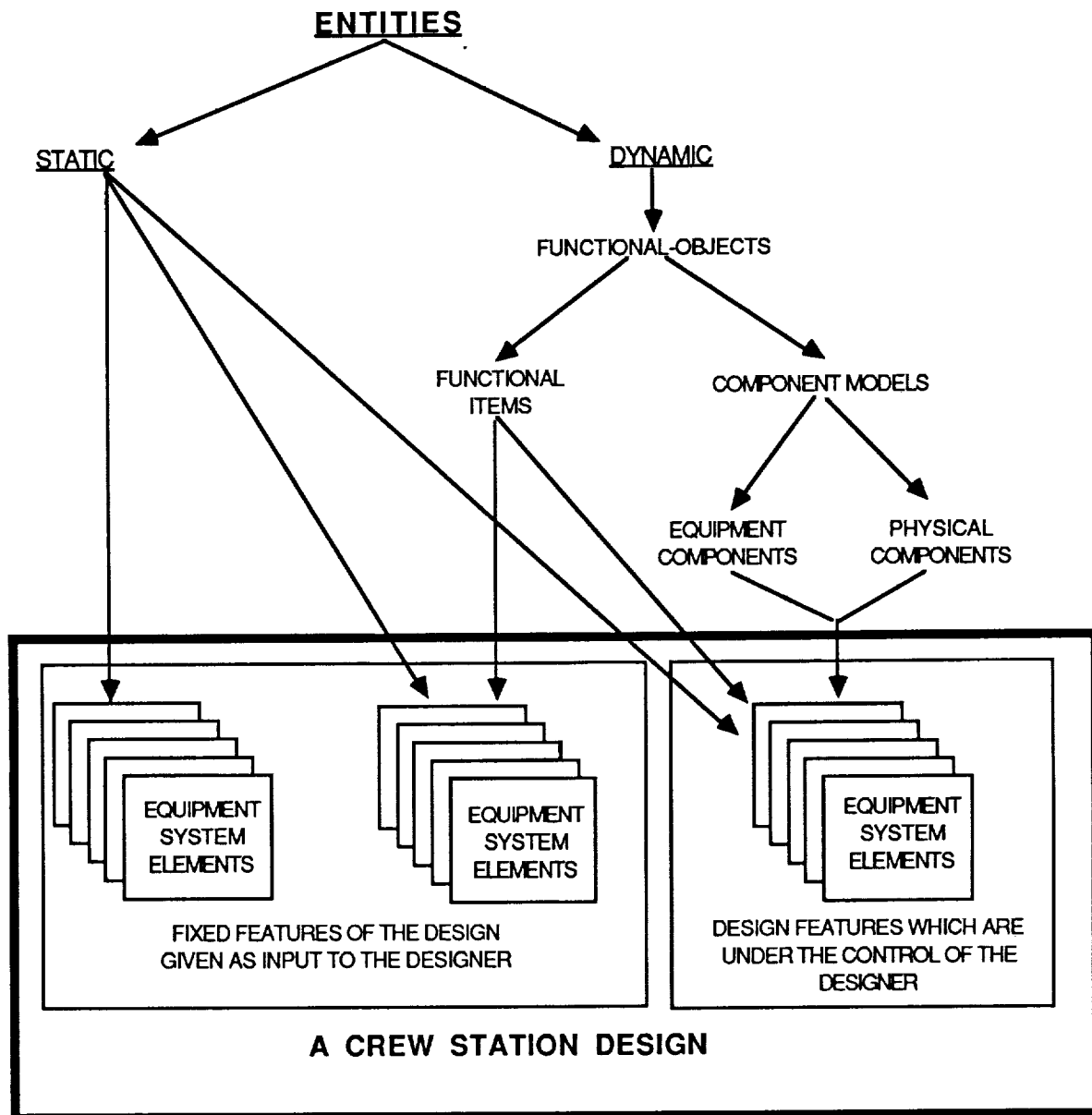


Figure 4.4 Equipment Modelling

4.4.2 Equipment Modelling Objectives

The objectives of the Equipment Modelling unit are directed at the needs of the designer in investigating different design alternatives. In some cases, the designer may be concerned with the functions provided by different systems and if these functions are sufficient for the given tasks. In other cases, the designer may be more concerned with the interaction between the pilot and the crew station when using a given system and focus more on the interface to a given function. In order to view equipment from the desired perspective, the designer needs flexibility in specifying either the functional or physical design attributes of a system. In order to provide this flexibility, the Equipment Modelling unit has the following objectives:

1. The capability to specify functional design attributes independent of physical design.
2. The capability to specify physical design attributes independent of functional design.
3. Provide a mechanism for integrating functional and physical designs into a single component.
4. Automatically determine the activities (physical interactions) necessary to achieve a given function based on attributes from both the functional and physical models.
5. Provide a component structure in which the functional or physical model elements may be conveniently varied.

4.4.3 Component Modelling

4.4.3.1 Component Modelling Overview

Components are represented in the Equipment Modelling unit by synthesis of functional and physical models. The synthesis is achieved by making values of attributes in both models dependent upon the other. The objects used to implement the models have been developed to facilitate accessing physical information from the functional models and functional details from the physical models in order to ensure flexibility for program development. Complete details are provided in the following sections.

4.4.3.2 Functional Components

4.4.3.2.1 Equipment Component Basic Flavor

EQUIPMENT-COMPONENT flavor

Defines flavor for EQUIPMENT-COMPONENT objects

The following local instance variables are used:

component-of----->instance which has this object
as a member of the list in the
components' slot.
component-name---->symbol indicating flavor of object
component-type----->user specified
component-functions---->list of COMPONENT-FUNCTION instances
physical-component---->instance of a PHYSICAL-COMPONENT
input-states----->specifies input-state descriptors
input-state-operators---->used in creating component-functions
output-states----->specifies output-state descriptors
activity-actions----->user specified
func-model-args----->holds values to be passed to a function
func-model-output----->holds value resulting from above
function.

The following instance variables are inherited from entity:

location
geometric-prop
functional-objects
functional-instances
functional-flag
current time
cycle-flag

(MAKE-INSTANCE EQUIPMENT-COMPONENT :AFTER) method

Purpose: Finishes initialization requirements and updates
values in both the equipment and physical models.
Args: Ignored
Called by: (MAKE-INSTANCE EQUIPMENT-COMPONENT)
Returns: N/A
Side-affects: 1. Sets value of (equip-component physical-component)
to the equipment-component instance. It is
necessary to do this in the AFTER method since
an instance cannot be passed to another instance
which is instantiated in the original instance's
make-instance method.
2. Maps physical component state-operators to
physical component states.
3. Determines component functions operators.

(UPDATE-INPUT-STATE-OPERATORS EQUIPMENT-COMPONENT) method

Purpose: Maps physical component state-operators to
equipment component states
Args: None
Called by: (MAKE-INSTANCE EQUIPMENT-COMPONENT :AFTER)
Returns: N/A
Side-affects: Sets values for input-state-operators.

(UPDATE-COMP-STATE-OPERS EQUIPMENT-COMPONENT) method

Purpose: Maps functions to operators which achieved states
enabling those functions
Args: None
Called by: (MAKE-INSTANCE EQUIPMENT-COMPONENT :AFTER)
Returns: N/A
Side-affects: Adds operators to component-functions.

(EVAL-FUNC-MODEL-ARGS EQUIPMENT-COMPONENT) method

Purpose: Evaluates functional model input arguments
Args: None
Called by: (PASS-FUNC-MODEL-ARGS EQUIPMENT-COMPONENT)
Returns: Returns list of evaluated arguments
Side-affects: None

(PASS-FUNC-MODEL-ARGS EQUIPMENT-COMPONENT) method

Purpose: Pass arguments to physical model
Args: None
Called by: Simulation models
Returns: N/A
Side-affects: Sets (func-model-args physical-component)
to world state sensitive arguments.

4.4.3.2.2 Equipment Component Example

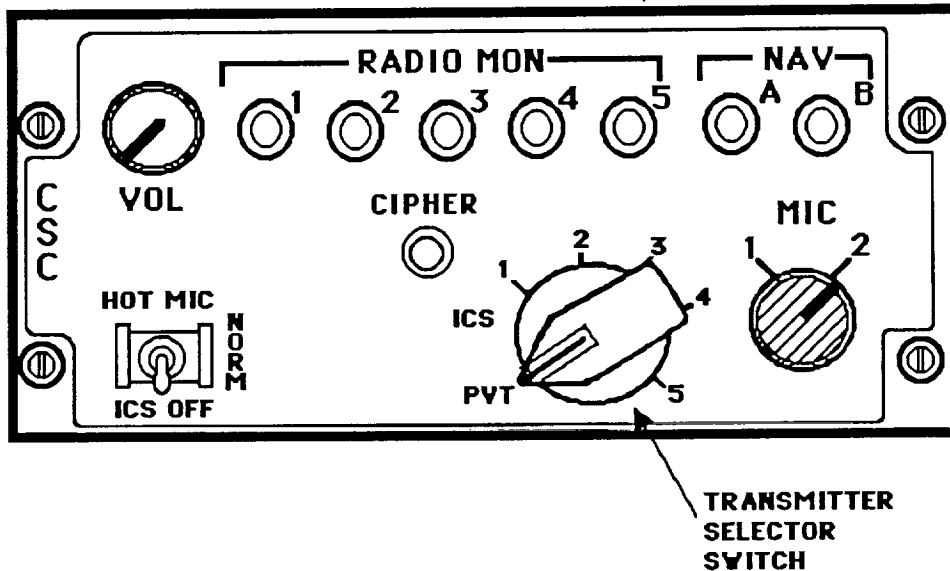


Figure 4.5 Transmitter Selector Switch

The Transmitter Selector Switch is located on the communication control panel and is used to select channels for intercommunication between the crew members and to select a radio transmitter for external communication.

C-1041-trans-selector flavor

The following local instance variables are used:

selector-state----->set during initialization and the simulation to the physical state value.
selected-transmitter--->set during initialization and the simulation.

The following instance variables are inherited from equipment-component

component-of----->should be set during instantiation
component-name----->set to 'C-1041-trans-selector in the default-init-plist
component-type----->user specified
component-functions---->detail discussion is presented in the following paragraphs.
physical-component----->set to the following in the default-init-plist
 (make-instance '7-position-discrete-rotation-control-1)
 :input-states '(selector-state)
 :output-states '(selected-transmitter)
input-states----->set to '(on/off-state volume-state) in the default-init-plist
input-state-operators---->set during instantiation.
output-states----->set to '(output-volume) in the default-init-plist
activity-actions----->user specified
func-model-args----->set to ((audio-signal pilots-VHF/FM) in the default-init-plist
func-model-output----->set during a simulation

The following IVs are inherited from entity through equipment-component:

name----->set to 'C-1041-trans-selector in the default-init-plist
entity-type----->set to 'functional in the default-init-plist
location----->N/A - this depends on the CDE.

The flavor has EQUIPMENT-COMPONENT as a flavor component.

The instance variable "component-functions" for the C-1041-trans-selector object is bound to a list of component-function objects. During flavor definition, the values for the ":function-pattern" and ":state-value" variable of the component-function instances are defined with the default-init-plist. As the C-1041-trans-selector object is instantiated, the values for the ":state-operator" variables are updated by the interaction of the make-instance "after" methods of both the C-1041-trans-selector object and the physical component object.

When the "after" methods are completed the "component-functions" variable is bound as follows:

component-functions =>

```
' ( <component-function-instance-1
: function-pattern '(connect (audio-signal microphone) (audio-input ICS))
: state-value '(= selector-state 1))
: state-operator

<component-function-instance-2
: function-pattern '(connect (audio-signal microphone) (audio-input ICS))
: state-value '(= selector-state 2))
: state-operator
```

```

      <component-function-instance-3
: function-pattern '(connect (audio-signal microphone) (audio-input pilots-ARC-186))
      :state-value '(= selector-state 3))
      :state-operator

      <component-function-instance-4
      :function-pattern '(connect (audio-signal microphone) (audio-input UHF-receiver))
      :state-value '(= selector-state 4))
      :state-operator

      <component-function-instance-5
: function-pattern '(connect (audio-signal microphone) (audio-input CPG-186))
      :state-value '(= selector-state 5))
      :state-operator

      <component-function-instance-6
      :function-pattern '(enables 'pilot's-remote-switch)
      :state-value '(= selector-state 7))
      :state-operator

```

4.4.3.3 Physical Components

4.4.3.3.1 Physical Component Basic Flavor

PHYSICAL-COMPONENT flavor

Defines flavor for PHYSICAL-COMPONENT objects

The following instance variables are used:

```

equip-component--->instance of EQUIPMENT-COMPONENT
input-states----->specifies input-state descriptors
state-operators--->actions which change state values
side-effects----->currently undeveloped but needed
func-model----->function for a functional model
func-model-args--->arguments for the functional model
installed-as----->user specified
installed-on----->user specified

```

The following instance variables are inherited from entity:

```

location
geometric-prop
functional-objects
functional-instances
functional-flag
current time
cycle-flag

```

Purpose:	Runs functional model
Args:	Passed internally previously. See (PASS-FUNC-MODEL-ARGS EQUIPMENT-COMPONENT)
Called by:	Simulation models
Returns:	N/A
Side-affects:	Sets func-model-output to the output of the functional model.

DISCRETE-ROTATION-CONTROL flavor

The following local instance variables are used:

- control-handle----->a symbol specifying control handle type
- rotation-state----->the current state
- rotation-states----->number of states available
- rotation-detent-style----->a symbol characterizing physical style
- rotation-travel----->specifies in degrees maximum amount of rotation possible
- rotation-travel-breakout---->number of degrees movement to reach the next state
- rotation-static-friction----->resistance to initial movement
- rotation-coulomb-friction--->resistance to continued movment
It not related to either velocity
or displacement
- increase-direction----->provides mapping for rotation direction to level values
- rotation-markings----->provides marking-to-state mappings

```

equip-component
input-states----->set in this flavor using the default
                      init plist to '(rotation-states)
state-operators
state-change-side-effects
func-model----->set in this flavor using the default
                   init plist to:
                   '(lambda (instance rotation-state)
                      rotation-state)

```

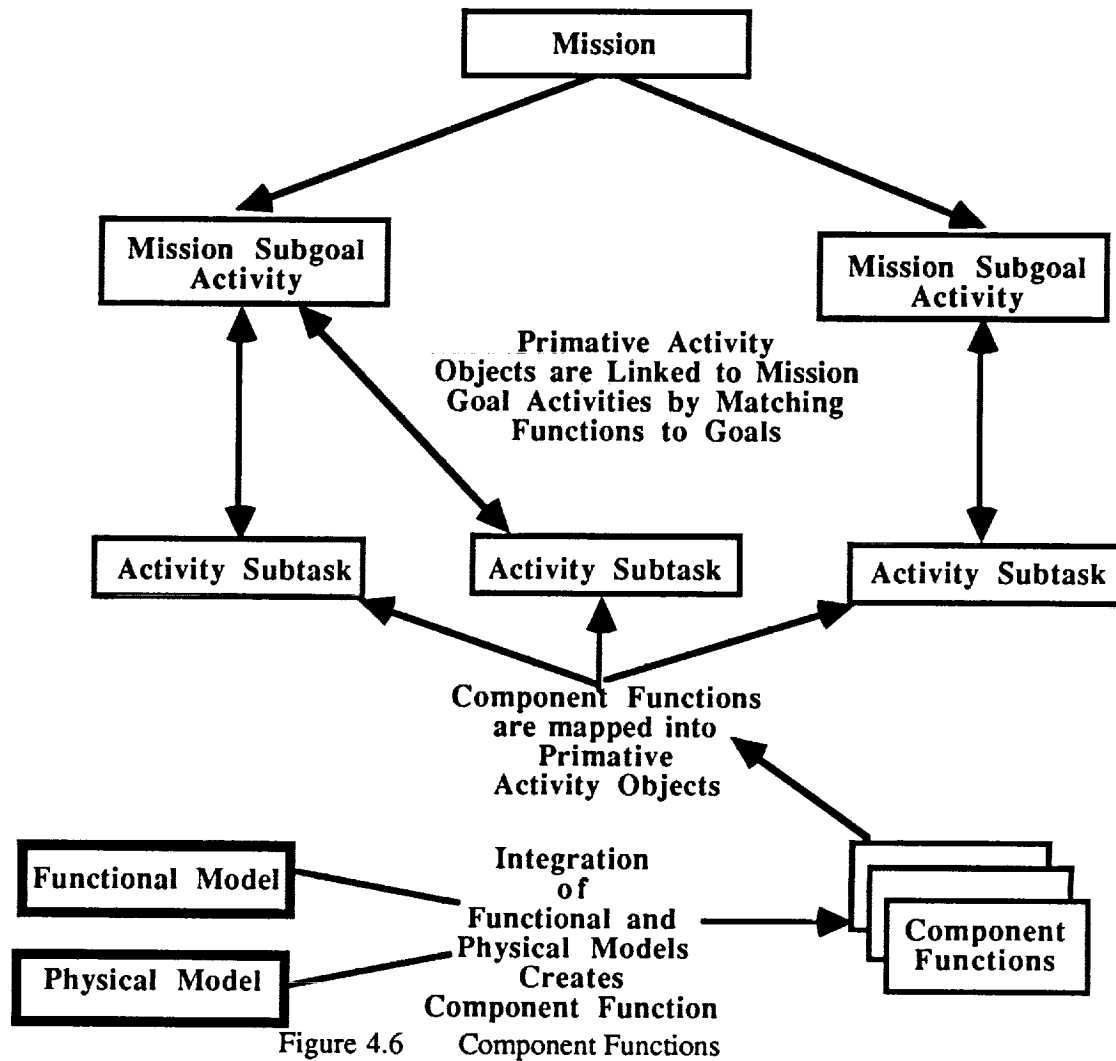
The following instance variables are inherited from entity:

- location
- geometric-prop
- functional-objects
- functional-instances
- functional-flag
- current time
- cycle-flag

4.4.3.4 Component-Functions

4.4.3.4.1 Component Functions Overview

Component functions are structures that provide an explicit link between the design of equipment and specific pilot subtasks. Component function values are dependent upon functional and physical design models and provide a flexible means of investigating different design alternatives. After a design has been chosen, it is necessary to create activities which map to component functions. This is currently done by hand in ZMACS but the structures have been developed with the intention of having the activities created automatically. These activities represent primitive actions which a pilot may perform and this process is essentially the "bottom-up" approach to task analysis. These activities need to be linked as subgoals for achieving higher level goal activities which are created in a "top-down" approach.



4.4.3.4.2 Component-Function Flavor

COMPONENT-FUNCTION flavor

Defines flavor for COMPONENT-FUNCTION basic objects

The following instance variables are used:

function-pattern----->specifies a function

state-value----->specifies a state which achieves the function

state-operators----->specifies operators which achieve the above state

The flavor is a base flavor and does not have any specialized flavor components

Component-functions objects are initially defined within the definition of the associated functional object. The instance variables "function-pattern" and "state-value" should be bound in the initial definition. The value for instance variable "state-operators" is set during instantiation and its value is dependent on both the functional object and a physical object to which the "physical-component" instance variable of the functional model is bound.

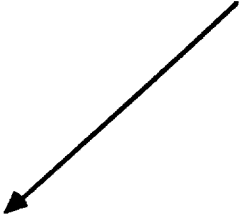
Components-functions objects are currently mapped directly into primitive activities using the ZMACS editor. Future developments should provide methods for generating activity object templates automatically based on value provided in the component- functions instances. It is not clear, however, at this time the range of attributes which will need to be specified for the activity in a process which generates activity instances from component functions.

4.4.3.4.3 Component Function Example

Example of a Component Function for a Transmitter Selector Switch

In the example, the Transmitter Selector Switch object has six functions as indicated in the component-function instance variable:

COMPONENT-FUNCTIONS: (#<Component-Function 54124226>
 #<Component-Function 54124232>
 #<Component-Function 54124236>
 #<Component-Function 54124242>
 #<Component-Function 54124246>
 #<Component-Function 54124252>)



#<Component-Function 54124236>, an object of flavor COMPONENT-FUNCTION
 has instance variables:

FUNCTION-PATTERN: (CONNECT (AUDIO-SIGNAL MICROPHONE)
 (AUDIO-INPUT PILOTS-ARC-186))
 STATE-VALUE: (= SELECTOR-STATE 3)
 STATE-OPERATORS: (DISCRETE-ROTATE-TO
 #<C-1041-TRANS-SELECTOR 54124176> 3)

→ The FUNCTION-PATTERN instance variable was set by the value supplied in default-init-plist of the Transmitter Selector Switch flavor.

→ The STATE-VALUE instance variable was set by the value supplied in default-init-plist of the Transmitter Selector Switch flavor.

→ The STATE-OPERATORS instance variable was set by the values computed in MAKE-INSTANCE "AFTER" method inherited by Transmitter Selector Switch flavor from the Equipment-Component flavor. This method integrates values from both the functional and physical objects to map a physical activity to a specific function.

4.4.5 Equipment Systems

4.4.5.1 Equipment Systems Overview

EQUIPMENT-SYSTEM flavor

Defines flavor for EQUIPMENT-SYSTEM objects

The following instance variables are used:

components----->flavor names of equipment components
which may be subsystems defined with
the EQUIPMENT-SYSTEM flavor.

related-equipment--->user specified

task-analysis-code-->user specified

subsystems----->component subsystems defined with the
EQUIPMENT-SYSTEM flavor.

operators-manual---->user specified

reference-manuals--->user manuals

The flavor is a base flavor and does not have any specialized flavor components

(FUNCTIONS-PROVIDED EQUIPMENT-SYSTEM) method

Purpose: Describes capability of system.

Args: None

Called by: As needed

Returns: A list of functions provided by the system in the
following format:

((<component-name-1> (<component-function-a>
<component-function-b>

..
<component-function-n>))

(<component-name-n> (<component-function-a>
<component-function-b>

..
<component-function-n>))

4.4.5.3 Equipment System Example

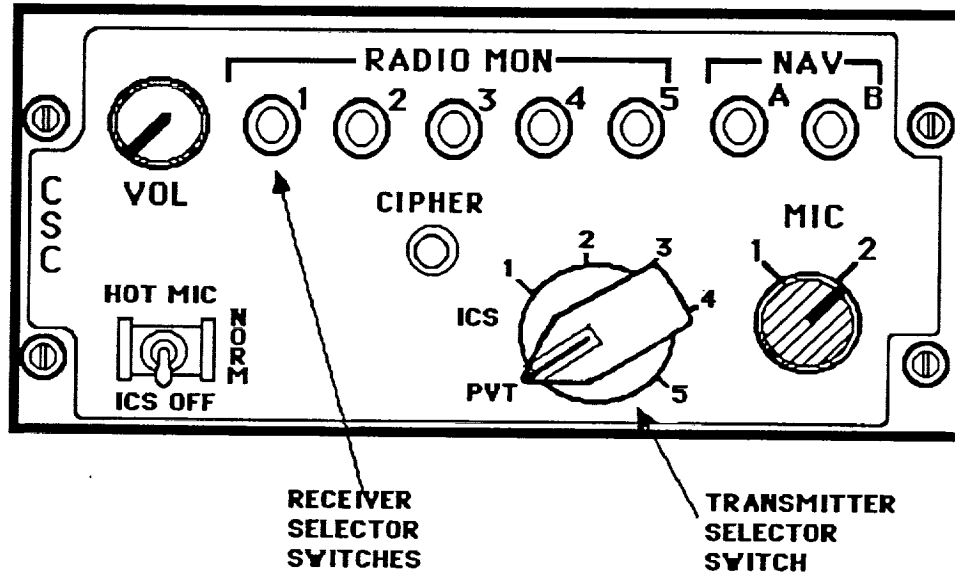


Figure 4.7 Communications Control Panel C-1041/ARC

Communications Control Panel, C-1041/ARC

C-1041 flavor

Defines flavor for C-1041 objects

The following local instance variables are bound as follows:

C-1041-volume-control	----->	instance of C-1041-volume-control
C-1041-rec-selector-1	----->	instance of C-1041-rec-selector-1
C-1041-rec-selector-2	----->	instance of C-1041-rec-selector-2
C-1041-rec-selector-3	----->	instance of C-1041-rec-selector-3
C-1041-rec-selector-4	----->	instance of C-1041-rec-selector-4
C-1041-rec-selector-5	----->	instance of C-1041-rec-selector-5
C-1041-rec-selector-AUX	-->	instance of C-1041-rec-selector-AUX
C-1041-rec-selector-NAV	-->	instance of C-1041-rec-selector-NAV
C-1041-hot-mike-switch	----->	instance of C-1041-hot-mike-switch
C-1041-trans-selector	----->	instance of C-1041-trans-selector

The following instance variables are inherited from functional-entity:

geometric-prop ----->dependent on the CDE models.
 functional-objects ---->set to a list of the following:
 C-1041-volume-control
 C-1041-rec-selector-1
 C-1041-rec-selector-2
 C-1041-rec-selector-3
 C-1041-rec-selector-5
 C-1041-rec-selector-AUX
 C-1041-rec-selector-NAV
 C-1041-hot-mike-switch
 C-1041-trans-selector
 functional-instances -->list of instances of functional-objects
 functional-flag ----->set to T

The following instance variables are inherited from DYNAMIC-ENTITY
 through FUNCTIONAL-ENTITY

current time ----->simulation time.
 cycle-flag ----->not currently used.

The following IVS are inherited from BASIC-ENTITY through
 DYNAMIC-ENTITY.

name ----->name of most specialized flavor component.
 entity-type----->Set to 'functional in the default-init-plist.
 location ----->this variable's structure is dependent on
 how we integrate with the CDE models.

The following instance variables are inherited from equipment-system:

components----->a list of the following:
 C-1041-volume-control
 C-1041-rec-selector-1
 C-1041-rec-selector-2
 C-1041-rec-selector-3
 C-1041-rec-selector-4
 C-1041-rec-selector-5
 C-1041-rec-selector-AUX
 C-1041-rec-selector-NAV
 C-1041-hot-mike-switch
 C-1041-trans-selector
 C-1041-MIC-switch
 C-1041-CIPHER-indicator
 C-1041-control-head-plate
 C-1041-hot-mike-switch-guard
 C-1041-mounting-screws
 C-1041-labels
 functions-provided--->a list of functional-instances functions
 the following format:
 ((<component-1> (<function-1>
 <function-2>
 :
 <function-n>)
 :
 (<component-n> (<function-1>

```

                                <function-2>
                                :
                                <function-n>)))
related-equipment---->a list of the following:
                                ICS pilots-ARC-186
                                UHF-receiver CPG-ARC-186
                                IFF-transponder
                                ADF
                                RADIO/ICS-rocker-switch
                                remote-trans-selector-switch
                                transmitter-lights-panel
                                headset microphone
task-analysis-code--->user specified as desired.
subsystems----->'none
operators-manual---->'TM 55-1520-238-10
reference-manuals--->'TM 55-1520-238-10
This flavor has the following flavor components:
                                functional-entity
                                equipment-system

```

4.4.5.2 Helicopter Modelling

4.4.5.2.1 Helicopter Modelling Overview

In order to model the basic characteristics of rotary wing flight in an unmanned simulation, it is necessary to model guidance logic and simple aerodynamic properties. Guidance logic and aerodynamics for rotary wing flight are complex topics and the requirements for models are dependent upon the tasks and the desired level of fidelity.

In Phase 1, guidance logic was represented as pilot activities in lisp code and the helicopter was represented as an lisp flavor object with the associated aerodynamics represented as lisp methods. Since the Phase 1 models were point-mass based which did not provide bank and pitch information necessary for the graphical models, Phase II provided models for aerodynamics and associated guidance in Fortran code. The aerodynamics model was linked to a lisp flavor object representing the helicopter by means of a communication program which passed information once during each tick interval. The pilot activities were linked to the guidance model by a methodology which assumed the aircraft was controlled by determining what control movements would be necessary to achieve a set of points specified as given values for X, Y, Z coordinates, heading, and airspeed. In this methodology, the pilot activities linked to the guidance model by passing a set of waypoints defined in terms of longitude, latitude, altitude, heading and airspeed. The guidance model is used to control the aircraft by providing the necessary control inputs to the aerodynamic model to achieve these waypoints. To model contour flight such as depicted in Figure 4.1, a route of flight would be defined and passed to the guidance routine as a set of waypoints representing points A through D. The guidance routine would determine the control movements necessary to fly through these points and pass them to the aerodynamic model which would update the helicopter state variables as the simulation progressed. In Phase III, the guidance and aerodynamic model code was moved from the Symbolics to the IRIS. The procedures for passing values from the Symbolic Modelling CSCI to the modules on the IRIS have not yet been resolved.

The aerodynamic model must also provide the symbolic models with state variable values for the above attributes.

4.5 Mission Modelling

4.5.1 Mission Overview

The mission description resides in the "mission" object of the LISP implementation. The mission provides the general description of the tasks which must be performed or states which should be achieved. This is the structure in which information normally provided by an operations order should be represented. This should not be confused with top-level mission activities since this structure should represent the actual mission and a top-level mission activity should represent the activity of trying to perform a mission. This object is referenced by the pilot model by the assignment of this object as the value of the mission variable of the pilot object. Information concerning the mission is accessed by message passing.

For example, a pilot may determine his current route by the following message:

```
(send (send self :mission) :current-route)
```

The indirect addressing of the above code provides flexibility since to change the pilot's mission the designer need only to change the value of the mission variable on the pilot's model object to a new mission object:

```
(send self :set-mission <new-mission-object>)
```

Similarly, value for the mission may be changed directly without the need to access multiple objects or structures. The simulation may develop in a way that the pilot is required to change his current route. This would be accomplished by the action of one of the pilot's activities sending the message:

```
(send mission :set-current-route <new-route>)
```

The separation of the mission object from the task representation is important in that it is the primary relationship which provides realistic task generation using a general task descriptions which are responsive to different situations.

4.5.2 The Mission Flavor

The mission defines the objectives for a pilot in terms of tasks which must be completed or states to be achieved and it should supply information necessary to achieve the objectives.

It is important to distinguish the mission from the top-level activity which represents an attempt to accomplish a mission. The mission flavors used in previous phases were hardcoded and not flexible for mission changes. Definitions of base mission flavors from which specific missions should be defined are not yet developed but are necessary for flexibility in defining different missions. The file "B:>p3>sym>mission>p2-mission.lisp contains information given to the

helicopter pilots in the Phase II A3I scenario by their mission and associated doctrine. It was not used in Phase III and is presented only as background.

AMBUSH-MISSION flavor

Defines flavor for AMBUSH-MISSION objects

The following instance variables are used:

```

name----->"Ambush Mission"
approach----->path of approach to observation area
observation-positions----->where to observe from
landmark----->position of landmark
fired?----->have I fired yet?
coordinated-firing?----->have we coordinated firing?
firing-positions----->planned firing positions
default-pop-up-elevation-->don't pop up higher than this
weapon-type----->what kind of weapon to use
missile-type----->what kind of missile to use
hover-mode----->manual or auto-hover preferred?
target----->what is my target?
target-order----->decreasing order of target importance
lead----->lead helicopter in mission
wing----->wing helicopter in mission

```

(:POP-APPROACH AMBUSH-MISSION) method

Purpose: Remove an approach point that has been reached.
 Args: None
 Called-by: Mission activities procedures
 Returns: N/A
 Side-effects: Sets approach instance variable to the cdr of itself.

(:NEXT-APPROACH-POINT AMBUSH-MISSION) method

Purpose: What is the next approach point to reach?
 Args: None
 Called-by: Mission activity procedures.
 Returns: N/A

(:WEAPON-TYPE? AMBUSH-MISSION) method

Purpose: What type of weapon should be used for this mission?
 Args: asker
 Called-by: Mission activity procedures.
 Returns: 'Missile, machine-gun or nil.

(:MISSILE-TYPE? AMBUSH-MISSION) method

Purpose: What type of missile should be used for this mission?
 Args: objective asker
 Called-by: Mission activity procedures.
 Returns: 'Tow, hellfire or nil.

(:HOVER-MODE? AMBUSH-MISSION) method

Purpose: What kind of hover should I use?
Args: None
Called-by: Mission activity procedures.
Returns: 'auto-hover

(:EXPOSURE-LIMIT AMBUSH-MISSION) method

Purpose: How long (maximum) should I stay exposed after my first firing?
Args: None
Called-by: Mission activity procedures.
Returns: 30 ;Note this value is hardcoded and insensitive to the situation.

(:PAST-EXPOSURE-LIMIT? AMBUSH-MISSION) method

Purpose: Have I been exposed too long since firing?
Args: None
Called-by: Mission activity procedures.
Returns: t or nil

(:TARGET-VALUE AMBUSH-MISSION) method

Purpose: Space invaders approach to ordering targets
Args: object
Called-by: Mission activity procedures.
Returns: Numerical value depending on target type.

(:LAST-TARGET AMBUSH-MISSION) method

Purpose: Given a list of legal targets, what is the last one in line?
Args: targets
Called-by: Mission activity procedures.
Returns: a target instance.

(:ALL-TARGETS? AMBUSH-MISSION) method

Purpose: What are all the targets of the mission?
Args: None
Called-by: Mission activity procedures.
Returns: A list of targets (list of convoy vehicles).

(:ROTATE-FIRING-POSITIONS AMBUSH-MISSION) method

Purpose: Put the first firing position at the end of the list
Args: None
Called-by: Mission activity procedures.
Returns: A rotated firing position list
Side-effects: Sets the first position in the firing position to the end.

(:SCAN-OBJECT-NUMBER? AMBUSH-MISSION) method

Purpose: How many targets do I need to see before I stop scanning?
 Args: None
 Called-by: Mission activity procedures.
 Returns: 4 ;hardcoded and not sensitive to the situation

(:MISSION-CHOICES AMBUSH-MISSION) method

Purpose: Normally, this would return mission doctrine activities, given the current situation. None have been added yet for the a3i demo.
 Args: None
 Called-by: Mission activity procedures.
 Returns: A list ;hardcoded as (list "JINK-AND-HIDE")

(:SCRIPT-ACTIVITIES AMBUSH-MISSION) method

Purpose: Normally, this would return activities mandated by script doctrine, but none have been added yet for the a3i demo.
 Args: None
 Called-by: Mission activity procedures.
 Returns: A list ;hardcoded as (list "RETURN-TO-BASE")

(:JINK-APPROACH AMBUSH-MISSION) method

Purpose: Which way should I go perpendicular to the missile coming at me? (I need to go distance in the jink, perpendicular to the missile, in a direction with a line of sight at my elevation. Assumption is that at least one of the bearings will work.)
 Args: agent missile distance
 Called-by: Mission activity procedures.
 Returns: (x y)

LEAD-MISSION flavor

Defines flavor for lead missions.

The following instance variable are defined.

name----->name of mission, such as, "Lead Mission"
 agent----->who is assigned this mission
 current-route----->route currently being taken
 initial-position----->starting position in x, y and z.
 route-to-op----->list of point defining route to observation point
 route-to-hp----->list of point defining route to
 route-to-fp1----->list of point defining route to firing point 1.
 route-to-fp2----->list of point defining route to firing point 2
 route-for-egress--->list of point defining route to return to base.
 target----->current target
 target-order----->list of target sorted by target value which is currently fixed.

The following variables are inherited from 'ambush-mission

name----->"Ambush Mission"
 approach----->path of approach to observation area
 observation-positions----->where to observe from
 landmark----->position of landmark
 fired?----->have I fired yet?
 coordinated-firing?----->have we coordinated firing?
 firing-positions----->planned firing positions
 default-pop-up-elevation-->don't pop up higher than this
 weapon-type----->what kind of weapon to use
 hover-mode----->manual or auto-hover preferred?
 target----->what is my target?
 target-order----->decreasing order of target importance
 lead----->lead helicopter in mission
 wing----->wing helicopter in mission

The flavor uses 'Ambush-mission as a flavor component.

(:HOLDING-POINTS LEAD-MISSION) method

Purpose: list (car (last route-to-hp))
 Args: None
 Called-by: Mission activity procedures.
 Returns: N/A

(:OBSERVATION-POINTS LEAD-MISSION) method

Purpose: Indicate place from which to observe area.
 Args: None
 Called-by: Mission activity procedures.
 Returns: Last position in route-to-op

(:FIRING-POINTS LEAD-MISSION) method

Purpose: Indicate firing positions
 Args: None
 Called-by: Mission activity procedures.
 Returns: Last positions defined in route-to-fp1 and route-to-fp2

(:RELEASE-POINT LEAD-MISSION) method

Purpose: Indicate release point
 Args: None
 Called-by: Mission activity procedures.
 Returns: Last position in route-for-egress

(:PLANNED-ROUTE LEAD-MISSION) method

Purpose: Indicates planned route of flight
 Args: None
 Called-by: Mission activity procedures.
 Returns: A list including the initial position and all positions defined in route-to-op, route-to-fp1, route-to-fp2 and route-for-egress.

(:TARGET-ORDER? LEAD-MISSION) method

Purpose: Sorts target according to their value.
 Args: targets

Called-by: Mission activity procedures.
Returns: A sorted target list

4.6 Pilot Modelling

4.6.1 Pilot Modelling Overview

There is structural correspondence between the crew's temporal division of goal-directed activities and the software activity spawning mechanism. For example, experienced aircrews tend to break a mission into phases of related activities. At a fairly high level of abstraction these are:

1. Enroute
2. Target Service
3. Egress.

Associated with these phases are particular types of tasks and doctrinal rules of operation. In decomposing the mission for simulation, tasks may be classified as belonging to one or another of these phases. In software from the primary activity of "ambush mission activity" spawns three children, "enroute", "target-service", "egress". These are constrained to be performed sequentially in nominal mission operation. This means, for instance, that the tasks associated with preparation must be satisfactorily completed before the next phase of tasks can be generated, or spawned. The software architecture thereby simulates the aircrew's cognitive decomposition of the mission into discrete phases.

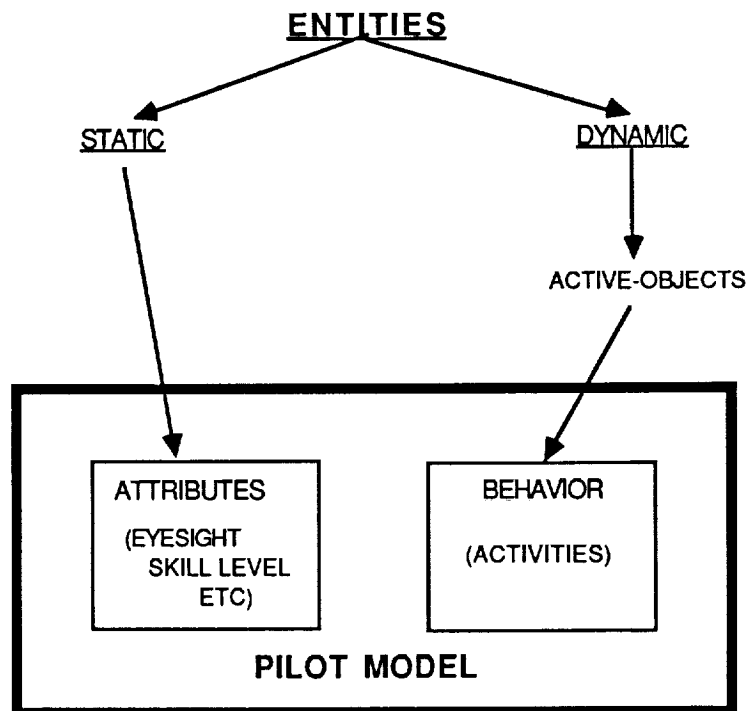


Figure 4.9 Pilot Model

4.6.2 Pilot Definition

4.6.2.1 The Pilot Basic Flavor

PILOT flavor

Defines flavor for pilot objects

The following instance variables are used:

helicopter----->instance of an helicopter
mission----->instance of an assigned mission

The following instance variables are inherited from ACTIVE-ENTITY

activities----->List of activities.
activity-history----->List of activities.

The following are inherited from DYNAMIC-ENTITY

current time ----->simulation time.
cycle-flag ----->not currently used.

The following are inherited from BASIC-ENTITY through DYNAMIC-ENTITY.

name ----->name of most specialized flavor component.
entity-t----->Set to 'active in the default-init-plist.
location----->This variable's structure is dependent on
how we integrate with the CDE models.

This flavor uses ACTIVE-ENTITY as a flavor component

4.6.3 Activity Representation

4.6.3.1 Activity Flavors

4.6.3.1.1 Test-Activity Flavor

TEST-ACTIVITY flavor

The following instance variables are used:

name----->name of the activity's flavor.
tag----->user specified.
agent----->who is carrying me out.
parent----->instance generating activity.
children----->instances generated by this activity.

activity-type----->activity flavor type.
 initialization-procedures-->to be completed when activity instantiated.
 time-init----->user specified.
 current-time----->simulation time.
 subactivities----->activities to be created.
 preconditions----->precondition for starting activity.
 estimated-start-time----->for planning.
 estimated-duration----->for planning.
 time-started----->when started as action.
 start-procedures----->to be completed when activity actually starts.
 tick-procedure----->form to execute in response to a tick message.
 time-ended----->time terminated. Nil = not terminated.
 termination-conditions---->when to end.
 termination-procedures---->what to do when I end.
 vacp-data----->hard-coded VACP value. See section 4.6.3.5.
 variable-vacp-load-p----->If nil, used hard-coded VACP value.
 If t, use result of evaluating vacp-load-form.
 vacp-load-form----->form which procudes context sensitive VACP values.
 vacp-data-history----->argument use with vacp-load-form recorded for
 debugging and analysis after the simulation.
 related-activities----->user specified.
 assertions----->known facts - user specified.
 constraints----->user specified
 heuristics----->user-specified

The flavor SI:PROPERTY-LIST-MIXIN is used as a flavor component.

(:INIT TEST-ACTIVITY :AFTER) method

Purpose: Enables before and after methods
 Args: Ignored
 Called-by: Procedures creating activities
 Returns: T
 Side-affects: Used to enable before and after methods only.

(MAKE-INSTANCE TEST-ACTIVITY :AFTER) method

Purpose: Creates children of an activity
 Args: Ignored
 Called by: Procedures for creating activities
 Returns: N/A
 Side-affects: Creates instances of activities listed in the subactivity
 instance variable and sets the children instance variable
 to a list of these instances. It also adds the activity to the
 global list name *test-activities* and names 'LEAD as the
 the agent. The naming of 'LEAD as agent should be changed
 to context sensitive code if the activity is for an environmental
 object.

(START TEST-ACTIVITY) method

Purpose: Starts an activity
 Args: time
 Called-by: Procedures for starting activities
 Returns: N/A
 Side-effects: Sets times and evaluates start procedures.

(TICK TEST-ACTIVITY) method

Purpose: Performs tick procedure if termination test fails.
 Args: None
 Called-by: Procedures in the activity's parent or, if top level, by the simulation executive.
 Returns: N/A
 Side-effects:

1. Updates current-time
2. Tests termination conditions
3. Terminates activity if terminations tests are true.
4. Executes tick-procedure otherwise.

(TERMINATE TEST-ACTIVITY) method

Purpose: Does the bookkeeping for completion of an activity.
 Args: None
 Called-by: Termination messages specified by the user or activity code.
 Returns: N/A
 Side-effects: Sets values for time and VACPs and executes termination procedures.

(TERMINATE? TEST-ACTIVITY) method

Purpose: Test for termination conditions within the context of the instance.
 Args: None
 Returns: None
 Side-effects: Test termination conditions.

4.6.3.1.2 Phase I and II Activity Flavors

Activity representation in previous phases were based on a methodology which assumed essentially a fixed scenario with simple contingencies. The methodology provided means of specifying a wide range of temporal relations between activities by providing the following activity flavors:

- Sequential
- Parallel
- Parallel-stop
- Rotation
- Fixed-duration
- Intermittent
- Choice
- Manual-choice

The temporal characteristics of an activity's subtasks were determined at compile time by providing them as flavor components. This enabled specific behavior by providing methods through inheritance but did not provide sensitivity to situations in which they would be performed since it was accomplished at run-time. Also, inherent in this methodology was the assumption that if goals

were temporally constrained then all the subtasks would inherit the temporal constraint. This assumption was shown to be in error in the communications example presented in Phase III in which it was incorrect to constrain the activity for reaching a switch for the second task until all subtasks for the first subtask were completed. It will be necessary to make major modifications to the activity mechanism to provide correct temporal constraints.

The activity flavors for Phases I and II provided significant capabilities for defining mission scenarios and, although the mechanism needs to be revised, the general functionality provided is still necessary. For this reason, a brief description of the activity flavors is provided. For more details concerning the actual mechanisms used, the documentation and source code for both Phases I and II should be used.

4.6.3.1.3 Sequential Activities

Sequential activities, when created, first executes its initialization procedures and then spawns a child from their first activity form. On termination of the activity spawned, they spawn a child from the next form, and so on, until all children have been spawned and terminated. The sequential activity then executes the termination procedures and terminates. Procedures included in the children spawned may be used to alter the sequence as a means of handling exceptions. For example, a sequential activity may include four activities as sequential forms and the first activity may have as its termination procedures a condition, which if proven true, will send a message to the parent activity to terminate and prevent the remaining three activities from being spawned.

4.6.3.1.4 Parallel and Parallel-stop Activities

Parallel and parallel-stop activities behave in a manner similar to sequential activities with a few exceptions. Parallel activities spawn all activity forms after initialization and the parallel activity continues until all the children have terminated. Parallel activities spawn all activity forms after initialization and the parallel activity continues until any of the children have terminated. To achieve state dependent behavior of terminating, a parallel activity may be used with the termination procedures of the children spawned having procedures which may terminate either another child or the parent activity.

4.6.3.1.5 Rotation Activities

Rotation activities spawn children from their activity form list in rotation until terminated. The test for termination may be present in the rotation activity, the activities spawned, or a combination of both.

4.6.3.1.6 Fixed Duration Activities

Fixed duration activities carry out their tick procedure for a fixed number of ticks, then terminate. The lowest level of activities are often fixed duration activities. The duration for fixed duration activities are specified in fixed duration lists. Representation of different level of training and experience may be represented by different fixed duration lists.

4.6.3.1.7 Intermittent Activities

Intermittent activities intermittently carry out a procedure. This is useful when trying to represent a cyclic activity for which the duration of execution and interval between activity is known.

4.6.3.1.8 Choice and Manual Choice Activities

Choice activities chooses one from a list of options spawned for the activity forms, then spawn that option after the time taken to choose it has elapsed.

An option for choice capability was added in Phase II that allows direct interaction with the designer. This enables the system to deal with decisions when the models involved do not yet possess sufficient capability to make a rational choice. This is accomplished by providing a choice menu to the designer which presents a description of the decision which must be made, details of factors which should be considered when making the decision, the choices available, and specific information relevant to each choice.

4.6.3.2 Complex Activities

Other types of activities may be created by blending activity types together or by procedures completed during activity initialization. Choice activities, for example, have the fixed duration activity flavor as a component. When they have decided which child to spawn, they set their duration to the amount of time spent choosing. On terminating, they add the child selected to their agent's activity list.

Another capability implemented in Phase II was the dynamic specification of activity forms to be spawned. In Phase I, activities to be spawned were specified prior to starting a simulation. As a simulation was run, arguments concerning the current world state were passed to these forms by the initialization procedures providing a means a making these predefined forms responsive to world state. In Phase II, the initialization procedures included functions which, when evaluated during a simulation, determined what activity forms would be spawned in addition to passing arguments concerning world state. This involved no changes to the basic structure of activities in Phase I, but simply involved providing a user defined function in the initialization procedures.

These activity types support some of the functionality one would desire in a simulation and planning environment. There are a tremendous number of obvious extensions. One strength of the object-oriented methodology is that it allows such extensions to be made without changing the existing system.

4.6.3.3 Activity Interactions with JACK

It is possible for activities to interact with the JACK CSCI by having the tick procedures of the activities pass commands to JACK. This is accomplished by having the activity determine which command, such as a "reach-for" command, it wants JACK to execute and having the Communications CSCI write this command to a file which the JACK program has been initialized to monitor. JACK would read the command, execute the action, and signal the activity when it has finished through the Communications CSCI. Since JACK has no concept of time, it is difficult to coordinate interaction of activities related to issues addressed by JACK.

4.6.3.4 Activity Interactions with Aero Modelling

At this time activity interactions with the Aero Modelling is limited to the interactions provided by the Guidance CSCI. Currently, these interactions are passive in that the activities can not vary the route of flight after the initial waypoints have been passed and the only interaction possible at run time is the ability to read state values of the aero model.

4.6.3.5 VACP Modeling

The operator performance model distinguishes among visual, auditory, cognitive, and psycho-motor resource loadings within a particular task. These loadings are expressed as subjective estimates of the workload imposed by activities as they are performed. The estimates have been supplied by expert pilots as they "walked" through the activities associated with a particular mission. The estimates are given on an integer scale from 1-7. The model assumes independent loading on each performance resource. The model further postulates an additive relationship along each of the resource dimensions.

In Phase I, the value for the specific VACP loads were determined by an activity table lookup, with different levels of loading provided to pilots of various experience and skill level by different tables. In Phase II, the capability to dynamically compute a load was demonstrated by having the value computed using world state details as factors. For example, in Phase II the "monitor radios" activity for adjusted its auditory load depending upon the amount of radio traffic which was determined by a random number function. This was implemented by providing all activities with a "variable-VACP-P" flag and "variable-VACP-form" values. The VACP load for an activity is determined by checking the value of the "variable-VACP-P" flag. If the value is nil, the value for the load is determined by the lookup table in the same manner as Phase I, otherwise the load is the result of evaluating the function provided as the "variable-VACP-form" in the lexical environment the activity was performed. This provides a simple framework for incorporating model computed loads as those models become available.

One important thing to note is that the VACP computation can be incrementally improved without damage to the system, since its functionality is independent of that of the rest of the system. Thus, as the model is made more and more adequate, the simulation environment may be used to test it without modification.

4.7 Task Decomposition

4.7.1 Task Decomposition Overview

The Task Decomposition Display was created specifically to demonstrate the concepts developed for Phase III concerning the relationship of the pilot's activities to specific equipment design features. It is intended to show how varying the design of crew station equipment affects a crew member's activities. The display provides a clear method of presenting which activities are affected by design and which are design independent.

This display was developed for the Phase III demonstration and will be revised during future development.

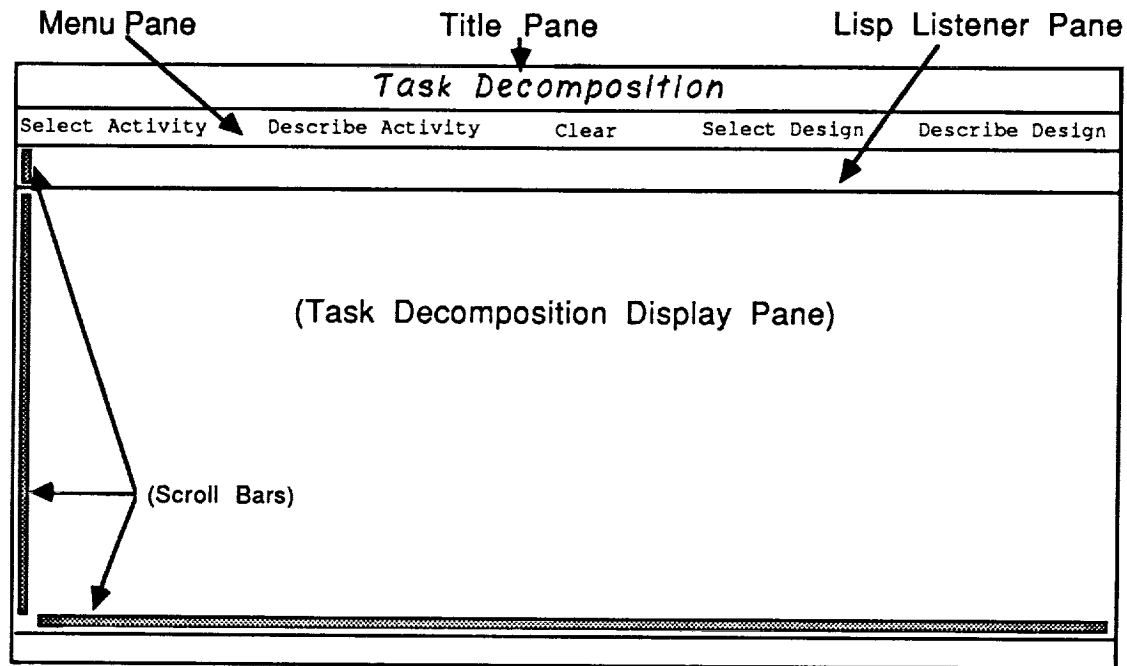


Figure 4.10 Task Decomposition Display

4.7.2 Task Decomposition Display

TASK-DECOMP-FRAME flavor

Defines a constraint frame flavor for TASK-DECOMP-FRAME objects

The following instance variables are used:

- td-title-pane
- td-mode-pane
- td-listener-pane
- td-display-pane
- select-activity-menu
- describe-activity-menu
- select-design-menu
- describe-design-menu

The following Symbolics provided flavors are used as components:

- tv:bordered-constraint-frame-with-shared-io-buffer
- tv:process-mixin
- tv:stream-mixin

(:INIT TASK-DECOMP-FRAME :BEFORE) method

Purpose: See side-effects and source code.
 Args: Ignored
 Called by: (:MAKE-INSTANCE TASK-DECOMP-FRAME)
 Returns: N/A
 Side-affects: Has the following side-effects:

1. Sets the global variable *task-decomp-frame* to the constraint frame instance.
2. Defines the following panes:
 - a. td-title-pane
 - b. td-mode-pane
 - c. td-listener-pane
 - d. td-display-pane
3. Creates a process for mouse handling,
4. Sets the configuration.

(:INIT TASK-DECOMP-FRAME :AFTER) method

Purpose: See side-effects and source code.
 Args: Ignored
 Called by: (:MAKE-INSTANCE TASK-DECOMP-FRAME)
 Returns: N/A
 Side-affects: Has the following side-effects:

1. Sets the global variable *task-decomp-frame* to the constraint frame instance.
2. Sets variables for the different panes.
3. Sets various attributes for panes.
4. Creates the following tv:momentary menus:
 - a. select-activity-menu
 - b. describe-activity-menu
 - c. select-design-menu
 - d. describe-design-menu

(:SELECT TASK-DECOMP-FRAME :AFTER) method

Purpose: Exposes constraint frame when it is selected.
 Args: Ignored
 Called by: System functions controlling window exposure.
 Returns: N/A
 Side-affects: Sends the instance an expose message.

(:REFRESH TASK-DECOMP-FRAME :AFTER) method

Purpose: Prints the title of the constraint frame in the title pane.
 Args: Ignored
 Called by: System functions controlling window exposure.
 Returns: N/A
 Side-affects: Invokes the function named "title" with the arguments "Task Decomposition" and td-title-pane.

(SELECT-ACTIVITY TASK-DECOMP-FRAME) method

Purpose: Enables the "select-activity-menu" tv:momentary-menu
Args: None
Called by: It is used as an constant by *TD-TOP-MENU* which is used to created menu item lists in the command pane of the display
 ;;; (td-mode-pane command-pane
 ;;; :default-character-style (:dutch :bold :normal)
 ;;; :item-list ,*td-top-menu*
Returns: N/A
Side-affects: Provides the following side-effects:
 1. Highlights select-activity in *TD-TOP-MENU*
 2. Determines position for the select-activity-menu.
 3. Sends an ":expose" message to the select-activity-menu.
 4. Sends a ":choose" message to the select-activity-menu.
 5. Sends a ":deactivate" message to the select-activity-menu.
 6. Removes highlight from select-activity in *TD-TOP-MENU*

(DESCRIBE-ACTIVITY TASK-DECOMP-FRAME) method

Purpose: Enables the "describe-activity-menu" tv:momentary-menu
Args: None
Called by: It is used as an constant by *TD-TOP-MENU* which is used to created menu item lists in the command pane of the display
 ;;; (td-mode-pane command-pane
 ;;; :default-character-style (:dutch :bold :normal)
 ;;; :item-list ,*td-top-menu*
Returns: N/A
Side-affects: Provides the following side-effects:
 1. Highlights describe-activity in *TD-TOP-MENU*
 2. Determines position for the describe-activity-menu.
 3. Sends an ":expose" message to the describe-activity-menu.
 4. Sends a ":choose" message to the describe-activity-menu.
 5. Sends a ":deactivate" message to the describe-activity-menu.
 6. Removes highlight from describe-activity in *TD-TOP-MENU*

(SELECT-DESIGN TASK-DECOMP-FRAME) method

Purpose: Enables the "select-design-menu" tv:momentary-menu
Args: None
Called by: It is used as an constant by *TD-TOP-MENU* which is used to created menu item lists in the command pane of the display
 ;;; (td-mode-pane command-pane
 ;;; :default-character-style (:dutch :bold :normal)
 ;;; :item-list ,*td-top-menu*
Returns: N/A
Side-affects: Provides the following side-effects:
 1. Highlights select-design in *TD-TOP-MENU*
 2. Determines position for the select-design-menu.
 3. Sends an ":expose" message to the select-design-menu.
 4. Sends a ":choose" message to the select-design-menu.

5. Sends a ":deactivate" message to the select-design-menu.
6. Removes highlight from select-design in *TD-TOP-MENU*

(DESCRIBE-DESIGN TASK-DECOMP-FRAME) method

Purpose: Enables the "describe-design-menu" tv:momentary-menu
Args: None
Called by: It is used as an constant by *TD-TOP-MENU* which is used to created menu item lists in the command pane of the display
 ;; (td-mode-pane command-pane
 ;; :default-character-style (:dutch :bold :normal)
 ;; :item-list ,*td-top-menu*
Returns: N/A
Side-affects: Provides the following side-effects:
 1. Highlights describe-design in *TD-TOP-MENU*
 2. Determines position for the describe-design-menu.
 3. Sends an ":expose" message to the describe-design-menu.
 4. Sends a ":choose" message to the describe-design-menu.
 5. Sends a ":deactivate" message to the describe-design-menu.
 6. Removes highlight from describe-design in *TD-TOP-MENU*

(MAKE-ACTIVITY-ITEMS TASK-DECOMP-FRAME) method

Purpose: Provides item list for the select-activity-menu
Args: None
Called by: (:INIT TASK-DECOMP-FRAME :BEFORE)
Returns: A list with the following format:
 ((<menu-item-name-1>
 :EVAL <form-1 to be evaluted>)
 .
 (<menu-item-name-n>
 :EVAL <form-n to be evaluted>))

(MAKE-ACTIVITY-DESC-ITEMS TASK-DECOMP-FRAME) method

Purpose: Provides item list for the describe-activity-menu
Args: None
Called by: (:INIT TASK-DECOMP-FRAME :BEFORE)
Returns: A list with the following format:
 ((<menu-item-name-1>
 :EVAL <form-1 to be evaluted>)
 .
 (<menu-item-name-n>
 :EVAL <form-n to be evaluted>))

(MAKE-DESIGN-ITEMS TASK-DECOMP-FRAME) method

Purpose: Provides item list for the select-design-menu
Args: None
Called by: (:INIT TASK-DECOMP-FRAME :BEFORE)
Returns: A list with the following format:
 ((<menu-item-name-1>
 :EVAL <form-1 to be evaluted>)
 .
 (<menu-item-name-n>
 :EVAL <form-n to be evaluted>))

(MAKE-DESIGN-DESC-ITEMS TASK-DECOMP-FRAME) method

Purpose: Provides item list for the select-design-menu
Args: None
Called by: (:INIT TASK-DECOMP-FRAME :BEFORE)
Returns: A list with the following format:
 ((<menu-item-name-1>
 :EVAL <form-1 to be evaluted>)
 .
 (<menu-item-name-n>
 :EVAL <form-n to be evaluted>))

TASK-DECOMP-FUNCTION method

Purpose: Old style of providing mouse tracking capabilities.
 It reads mouse blips and determines what action should be taken.
Args: A window instance.
Called by: This function is invoked in a separate process by the method (:INIT TASK-DECOMP-FRAME :BEFORE) when initializing the the display's constraint frame and runs continuously. When it is called, a separate process is created.
Example:
 The tv:process variable from the tv:process-mixin flavor component is set to the following value:
 tv:process '(task-decomp-function :special-pdl-size 4000 :regular-pdl-size 10000)

Returns: N/A

Side-affects: Mouse handling capabilities.

Note: *****
 ** NOTE **
 ** This is the old style of providing mouse handling capabilities **
 ** and should be replaced with Presentation system functions **
 ** *****

MAKE-TASK-DECOMP-FRAME function

Purpose: Makes task decomposition display frame
Args: None
Called by: Initialization functions as needed
Returns: *task-decomp-frame* bound to a task-decomp constraint frame display
Side-affects: If *task-decomp-frame* bound, kills previous value.

The following Symbolics function is used to assign a select key to access the task decomposition display. This function is invoked when the file is loaded.

Task-decomp Mode = Select Symbol-Shift-a

```
(tv:add-select-key #\ 'task-decomp-frame "Task-decomp Mode"
  '(send *task-decomp-frame* :select))
```

4.8 Mission Simulation

4.8.1 Overview

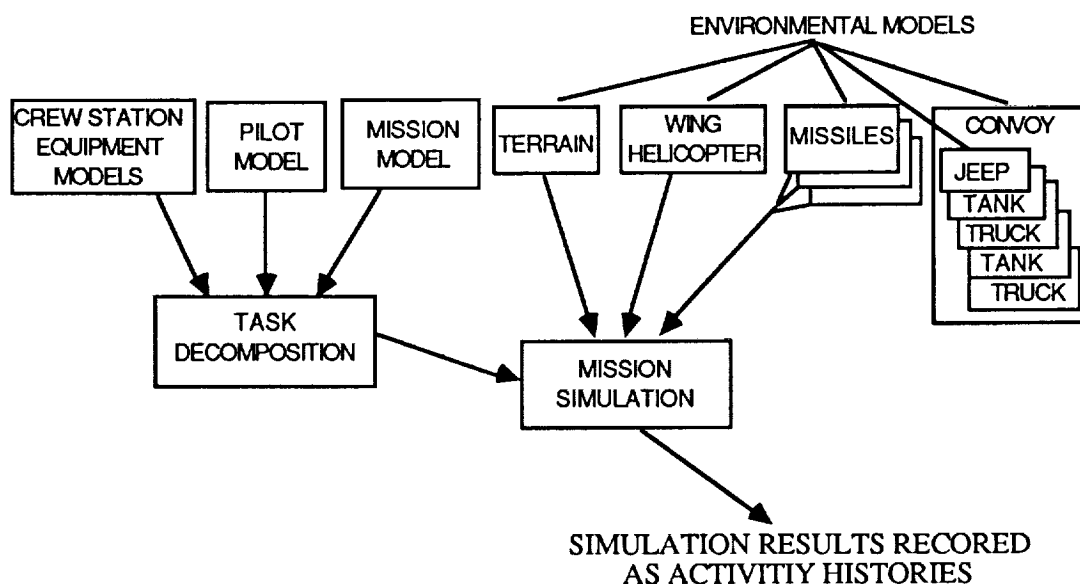


Figure 4.11 Mission Simulation

4.8.2 Simulation Requirements

The Symbolic Modelling CSCI produces objects which are used as input to the Simulation CSCI. One of the objects, the pilot model, has a task decomposition represented as a list of activities bound to the "activities" instance variable. These activities are structured in a manner to provide a driving script for the pilot's behavior during the simulation. Other objects are available to the simulation as active objects which respond in a manner similar to the pilot model object or as functional objects which respond not as result of having activities but rather as functional models responding to state changes. As explained previously, the simulation module runs a simulation by sending "TICK" messages to selected simulation objects. These objects, in turn, perform whatever actions are appropriate for the duration of time that a "TICK" represents. When all objects have completed processing, then the simulation issues another "TICK" message and the cycle continues.

5.0 NOTES

5.1 Miscellaneous

A significant portion of the code developed in Phase III was created simply to develop the concept of separate functional and physical models for each component. The structures used were guided primarily by an evolving process and were intended only as prototype code to convey concepts. It was never the intent that this code would represent basic structures which should be used as the underlying structures for development in future phases.

5.2 Limitations

Many of the concepts developed in Phase III assumed the availability of mechanisms developed in previous phases but did not explicitly use them. In fact, many of the mechanisms from previous phases are incompatible with code developed in Phase III.

This is especially true in the area of displays. Most of problems concern low level Lisp implementation issues and not the underlying concepts. Programmers should be aware of the necessity to at least review, if not, rewrite previously developed code.

5.3 Future Directions

5.3.1 Activity Scheduling

Activity scheduling in previous phases has been accomplished by compile-time decisions as to the temporal order of activity sequencing. The mechanism used was restrictive and often resulted in artificial nodes being created in an activity hierarchy simply to allow simple mixed temporal relations between subactivities. Also, errors had been noticed in the assumption that temporal constraints for goals will always be passed to all of the goal's subtasks. The temporal intervals proposed by James F. Allen (See section 2.1) provide significant improvements for representing temporal relations of activities and should be investigated further. It is important to observe that sequencing of the activities have been accomplished by control spread across the activity tree and internal to the activities themselves. The impact of external mechanism controlling the sequencing of activities has not been estimated at this time.

5.3.2 Decision Modelling

A method for decision modelling was provided in Phase 1 for decision modelling. During Phase II, although the method was still functional it was not utilized since the phase was focussing on other areas. In Phase III, there was no attempt at decision modelling and some basic structures concerning activities were changed. These changes will require that the decision modelling method be revised, however, these revisions are basically at the lisp implementation level and the concepts would essentially be the same. Changes proposed for activity scheduling may have significant impact on the conceptual structure needed by the Phase I decision modelling method. The following paragraph is taken from the Phase 1 documentation to provide a sample of the concepts of the Phase 1 decision modelling. For detail information the reader is referred to the Phase 1 documentation.

There is a functional correspondence between the aircrew's division of tasks into "Management Categories", and the way the software selects tasks to be performed. In addition to structuring tasks by phase, or mission segment, experienced aircrews cluster tasks according to the purposes they serve. For example, and again at a fairly high level of abstraction, tasks are divided as those concerned with flight, those associated with actual mission performance, and those in support of maintaining both the aircraft's and the mission's integrity. The software structure captures this functional distinction by providing a "slot" in the characterization of activities that weights the importance of the activity (and consequently the likelihood that it will be selected to be performed) based on the function it serves. The ranking parallels the aircrew's concern for flight being of primary importance, mission accomplishments next, and support activities third.

5.3.3 Aircraft Guidance

The requirement for defining the x, y, z, heading, and airspeed for each point in a route makes it extremely difficult to define many normal flight maneuvers. It will be necessary to make significant changes in the guidance module in order to model typical tactical flight scenarios.

5.3.4 Function Allocation

The methodology developed in Phase III for integrating functional and physical models of a component seem appropriate for investigating methodologies for representing the process of function allocation during the conceptual design phase.

5.3.5 Mission Modelling

Currently the mission object and the top-level mission activity are defined separately. It is important that the relationships between the information normally provided in a operations briefing (the mission object) and the pilot's interpretation of a mission (the top-level mission activity) to be represented explicitly. At this time, the distinction between the actual mission requirements and an attempt to perform a mission is unclear and as a result it is not clear by which standard the performance of a mission should be evaluated.

Annex B

Army-NASA Aircrew/Aircraft Integration Program

A³I

**Software Detailed Design Document:
View**

prepared by

Andrew Lui

December 1988

Table of Contents

1.0 INTRODUCTION.....	B-1
1.1 Identification	B-1
1.2 Scope.....	B-1
1.3 Purpose	B-1
2.0 RELATED DOCUMENTATION.....	B-1
2.1 Applicable Documents	B-2
2.2 Information Documents.....	B-2
3.0 ENVIRONMENTS AND DESIGN APPROACH	B-2
3.1 Requirements and Rationale.....	B-2
3.2 Hardware Description.....	B-3
3.3 Software Environment.....	B-4
3.3.1 Interface with the Operating System.....	B-4
3.3.2 Interface with Other Software Components.....	B-4
4.0 DETAILED DESIGN DESCRIPTION	B-4
4.1 Organization.....	B-4
4.2 Unit Detailed Design	B-7
4.2.1 Interface Manager Subsystem (IM).....	B-7
4.2.1.1 The IM data structures: imstrucs.h	B-7
4.2.1.2 The Window Manager: imwind.c.....	B-8
4.2.2 MultiGen Kernel Subsystem (KER)	B-8
4.2.3 Data Base Logic Subsystem (DBL)	B-8
4.2.3.1 The Flight Data Base data structures: fltfmt.h.....	B-9
4.2.3.2 The Flight Data Base color routines: fltcolor.c.....	B-10
4.2.3.3 The Flight Data Base logic routines: fltdbl.c	B-10
4.2.3.4 The Flight Data Base input an output routines: fltio.c.....	B-11
4.2.3.5 The linkage between the MultiGen Kernel and the Flight Data Base logic routines: fltlink.c	B-12
4.2.4 Animation Kernel Subsystem (ANA).....	B-12
4.2.4.1 The Animation control module: animate.c	B-14
4.2.4.1.1 Initialize Animation menu to the MultiGen system: ANA_init ().....	B-14
4.2.4.1.2 Control of ANA pulldown menu selection: Anamenu ().....	B-14
4.2.4.1.3 Get the animation set up file name: Get_filename ()	B-15
4.2.4.1.4 Read the content of the set up file: Get_setup ().....	B-15
4.2.4.1.5 Echo the content of the set up file: Setupanimate ()	B-15
4.2.4.1.6 Check the validity of the world objects: Checkbead ().....	B-15
4.2.4.1.7 Finish the set up process: Setupdone ()	B-16
4.2.4.1.8 Open the world view window: Set_world ().....	B-16
4.2.4.1.9 Open the moving view window: Set_copter1 ().....	B-16
4.2.4.1.10 Do transformation on object for world view: ANA_transformation ().....	B-17
4.2.4.1.11 Do transformation on the object for pilot view: ANA_pilotview ()	B-17
4.2.4.1.12 Form transformation matrix for sensor's view: Sensorview ().....	B-17

Table of Contents

4.2.4.1.13 Form transformation matrix for sensor's view: Observerview ()	B-18
4.2.4.1.14 Form transformation matrix for fixed camera view: Cameraview ().....	B-18
4.2.4.1.15 Prepare for animation: ANA_ethernet_ready ().....	B-18
4.2.4.1.16 Ready to read animation data: ANA_communicate ().....	B-18
4.2.4.1.17 Draw all windows: ANA_draw_pictures ()	B-18
4.2.4.2 The Flight path module: path.c.....	B-18
4.2.4.2.1 Define the flight path: PAT_define_path ().....	B-18
4.2.4.2.2 Save user defined flight path: PAT_done ().....	B-19
4.2.4.2.3 Locate the terrain surface: PAT_trackplane ().....	B-19
4.2.4.2.4 Fly Through the flight path: PAT_fly_path ()	B-19
4.2.4.2.5 Draw the flight path: PAT_draw_flight_path ()	B-19
4.2.4.3 Read simulation data module: simdata.c	B-19
4.2.4.3.1 Read data from a standalone file: SIM_read_standalone ()	B-19
4.2.4.3.2 Read data from pipe file: SIM_mess2data ().....	B-19
4.2.4.3.3 Terrain location adjustment: Hi_res_adjustment ().....	B-19
4.2.5 DMA Kernel Subsystem (TER).....	B-20
4.2.5.1 The standalone preprocessor module: readdma.c.....	B-20
4.2.5.2 Converting elevation data into polygonal surfaces.....	B-23
4.2.5.2.1 Read post file module: terdma.c.....	B-23
4.2.5.2.2 DMA user interface module: terrain.c.....	B-24
4.2.5.2.3 Generate terrain polygons module: terpoly.c	B-24
5.0 NOTES	B-24
5.1 Miscellaneous	B-24
5.2 Limitations.....	B-24
5.3 Future Directions	B-25
6.0 USERS GUIDE	B-25
6.1 Introduction	B-25
6.2 Related Documentation.....	B-25
6.3 Overview of Purpose and Functions	B-25
6.4 Installation.....	B-25
6.4.1 Installing MultiGen Software Environment.....	B-25
6.5 Start-up and Termination	B-26
6.6 Functions and Their operation	B-27
6.6.1 Setting up for Phase III demonstration	B-27
6.6.2 Creating model	B-28
6.6.3 The Animation menu.....	B-28
6.6.4 The Terrain menu	B-31
7.0 APPENDICES	B-31
7.1 Glossary, Abbreviations	B-31
Appendix A The content of makefile for creating MultiGen executable.....	BA-1
Appendix B The content of a script file for extracting data records from DMA's DTED distribution tape	BB-1
Appendix C The content of a script file for concatenate the disk files into a single tape image disk file	BC-1
Appendix D message.h	BD-1

Table of Contents

Appendix E	animate.menu.....	BE-1
Appendix F	animate.msg.....	BF-1

1.0 INTRODUCTION

1.1 Identification

This document establishes the requirements and detailed design of the Views Computer Software Configuration Item (CSCI), which forms a part of the A³I Computer Program System. Descriptions of the detailed processing requirements, structure, I/O, and control are provided for each lower level Computer Software Component (CSC), units, or function contained within the CSCI.

1.2 Scope

This document describes the functions, composition and the use of the Views software completed for Phase III demonstration of the A³I simulation. This document assumes that the reader is familiar with computer graphics concepts, computer animation, 3D geometric modeling techniques, window oriented user interface, C programming language, UNIX operating system, Silicon Graphics Inc.'s Graphics Library, and the internal structure of MultiGen if modification of the source code is required.

1.3 Purpose

The purpose of the Views software is to generate a set of tools for creating the A3I simulation world. The adopted software package for the Views software is called MultiGen which is developed by a company called Software Systems. MultiGen is a modeling system for creating and editing of three dimensional graphics data bases. The software is written in C and extensively modified by the A³I staff for the needs of the A³I simulation. The tools developed by A³I staffs are:

- (i) a capability to extract elevation data from any standard Level 1 Defense Mapping Agency's (DMA) Digital Terrain Elevation Data (DTED) tapes and convert into a shaded 3D graphics polygonal representation of terrain surface.
- (ii) a capability to create the world objects, such as tanks, trucks, helicopters or any objects to be included within the simulation world. Also, values of these world object's dynamics parameters can be assigned, so that their behaviors can be monitored during simulation.
- (iii) a capability to display the whole simulation in different viewing perspectives, so that the "big picture" of the simulation can be gleaned. It is intended to help a designer to get a good high-level view in near-real-time display of shaded graphics in which the simulation is proceeding. Windows are displayed on the screen to show the simulation. Each window represents a unique eye point location to monitor the simulation - such as the god's eye view to look at the whole gaming area at a glance, an observer's view inside the gaming area to monitor the movement of the convoys vehicles, a pilot's view to show what the pilot could see at his position inside the cockpit.

2.0 RELATED DOCUMENTATION

2.1 Applicable Documents

Software Systems MultiGen - Modeler's Guide, Interface Manager, MultiGen Kernel, Writing MultiGen DBL, Release 3.0, Software Systems, San Jose, September 1988.

Data Format Specification for Software Systems Flight Data Bases, Format Release 4, July 28, 1988.

Software Systems MultiGen, DMA Terrain Conversion Option User's Guide, (July, 1988).

Andrew P. Lui, *Phase II Views Software - Software Component Description Documents for A³I Program*, 1988.

2.2 Information Documents

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1978.

Marc J. Rochkind, *Advanced UNIX Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1985.

J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Massachusetts, 1982.

Nadia Magnenat-Thalmann and Danial Thalmann, *Computer Animation, Theory and Practice*, Springer-Verlag, Tokyo, 1985.

Silicon Graphics Inc., *IRIS User's Guide*, Volume I and II, Version 3.0, Mountain View, California, 1986.

Silicon Graphics Inc., *IRIS Programmer's Manual, Volume IB, System Calls and Subroutines*, Version 2.1, Mountain View, California, 1986.

DMA Aerospace Center, *Defense Mapping Agency (DMA) Product Specifications for Digital Terrain Elevation Data (DTED)*, PS/ICD/200, PS/ICF/200, Second Edition, St. Louis, Missouri, April 1986.

3.0 ENVIRONMENTS AND DESIGN APPROACH

3.1 Requirements and Rationale

The Views CSCI of the A³I Phase III software was developed with at least these goals in mind:

- To rapidly create and/or edit 3D geometric models of simulation objects.
- To provide a high-level, intuitive look at the progression of the Mission Simulation for the cockpit designer and mission analyst.

- To provide, at a cost well below that of visual simulation systems and at a speed that was not necessarily real-time but at least interactive, a 3D display of the events of the simulation from continuously variable viewpoints.
- To enhance the 3D viewing, not with excessive levels of detail, but with graphic cues sometimes more informative than the bird's-eye view of events.
- To incorporate interactive graphic tools for constructing and displaying, with a minimum of difficulty, a 3D world sufficient to portray the Mission Simulation.
- To provide a convenient interface capability for integrating the 3D graphics system with the simulation system.
- To avoid the high cost of developing much of the 3D graphics and geometric modeling software by buying and integrating as much of it as possible.

The aspects of the IRIS 2500T important to Phase III development were a fast graphics engine in an inexpensive workstation; an extensive library to exercise the engine; a UNIX program development environment, with tools that are widely used and that therefore require little learning for those who have used a UNIX system before; and a relatively powerful CPU for the price (a 68020-series processor with floating-point accelerator).

MultiGen was intended to provide both a means of constructing and editing the geometric objects of the 3D world, and a library of tools for displaying those objects based on instructions provided by the Mission Simulation.

3.2 Hardware Description

The IRIS 2500T as configured for this project contains a 68020 central processing unit, a floating-point accelerator for the CPU, 12-megabyte program memory, a frame buffer or image memory of 1024x1024x32 bits, a geometry engine for 3D coordinate transformations, a proprietary microcoded display processor and frame-buffer controller, a 19-inch 60 Hz non-interlaced 1024x768 RGB color monitor, an Ethernet interface with TCP/IP and NFS softwares, a keyboard and a mouse for user input, and two 440-megabyte disk drives.

Of the above mentioned hardware, the most important one is the combination of the geometry engine and display controller in the graphics pipeline which provide high speed rendering and fast 3D transformation into 2D images. Several other elements are also important too. The Ethernet controller makes possible the interfacing of a Symbolics 36XX and the IRIS without hardware development. The integration of the IRIS graphics library, with input devices, including a mouse, a button box and a dials box, make it possible to use these devices easily by calling library functions only. Finally, the large disk drives have proved useful for storing sizable quantities of graphics data and code.

The Views/MultiGen software is not limited to IRIS 2500T machine only. Basically, the Views/MultiGen can be run across the product line designed by Silicon Graphics Inc. For example, the Views/MultiGen software can be run in the IRIS 4D/GT machine if all the source code is compiled properly. Chapter 6 describes how to install Views/MultiGen in the machine.

In addition to the 2500T, a HP VT100 emulation terminal is used to aid in debugging by allowing source code to be displayed while graphic images are on the IRIS.

3.3 Software Environment

The most important elements of the IRIS 2500T software environment to the Phase III of A³I Views software are :

- A general-purpose, easy-to-use graphics library, the IRIS Graphics Library II (GL2).
- A standard Ethernet interface protocol (TCP/IP).
- The UNIX V operating system with Berkeley (4.3 bsd) extensions. C compiler, a development/debugging environment integrated with the C compiler and UNIX tools.
- The Software Systems' Interface Manager which provides a mouse and window oriented user interface on the IRIS workstation. This type of interface was originally developed by The Xerox Palo Alto Research Center (PARC) and has been popularized by the Apple Macintosh computer. MultiGen is written under the Interface Manager environment in which commands are typically selected from a pull-down menu by a cursor positioned with a mouse.

3.3.1 Interface with the Operating System

The Views software makes use of system calls which invoke UNIX primitive operations and other system subroutines and libraries. These functions are documented in Sections 2 and 3 of the *UNIX Programmer's Manual, Volume IB: System Calls and Subroutines*.

3.3.2 Interface with Other Software Components

The animation portion of the Views software can be operated in two different modes -standalone mode and communication mode. In standalone mode, the Views software gets the simulation data from a data file. In communication mode, the Views software communicates only with the Executive which is also running on the same IRIS 2500T. The Executive sends simulation data to Views software through a named pipe file (or FIFO). Detail description on this inter-process communication feature can be found in a book called *Advanced UNIX Programming* by Marc J. Rochkind (Prentice-Hall, 1985). The Views software uses the "read" system call to get all the simulation data from the pipe file, then animates the simulation objects (helicopters, convoy vehicles and missiles) in a 3D graphics world that consists of the gaming area. The content, format and commands for this communication are described in the Phase II document on the A³I Executive Module.

4.0 DETAILED DESIGN DESCRIPTION

4.1 Organization

The Views software is integrated within the MultiGen system. MultiGen is an interactive modeling system for creating, editing, and viewing 3D data bases for visual simulation. The original

MultiGen system is composed of three separate software subsystems—the Interface Manager Subsystem, the MultiGen Kernel Subsystem and the Data Base Logic Subsystem which are linked together to make an executable MultiGen program. On top of the original MultiGen system, the A³I staff created two additional software subsystems for our simulation requirements. The two newly added subsystems are the Animation Kernel Subsystem and the DMA Kernel Subsystem. The MultiGen system is implemented on the Silicon Graphics Workstation Iris 2500T. The Graphics Library of the IRIS 2500T is a set of graphics and utility routines that provide high/low level support for graphics. Figure 1 shows the name of each source code file and how they are organized in Views software.

- The Interface Manager Subsystem is a collection of procedures that support a mouse and window oriented user interface. This subsystem accepts all the commands from the user and passes to the appropriate subsystem.
- The MultiGen Kernel Subsystem consists of a programming environment and a graphics editor. The programming environment provides procedures that can be called to create or manipulate MultiGen's internal format and parts of the user interface, and to perform other utility functions.
- The Data Base Logic Subsystem (DBL) provides data base independence in MultiGen. Its functions are to access an "ofmt", or original format, for the MultiGen Kernel to handle user interface issues such as coordinate displays, terminology and modes, and to provide special purpose menu functions for an "ofmt". It is implemented as a series of low level procedures that are called from MultiGen Kernel via the DBL linkage.
- The Animation Kernel Subsystem provides a facility to view the A³I Mission Simulation in different viewing perspectives using 3D graphics. Windows are displayed on the screen to present the views. The Animation Kernel Subsystem obtains simulation data from the Executive program through inter-process communication pipeline or reading from a data file.
- The DMA Kernel Subsystem is used to extract elevation data from any standard Level 1 DMA DTED tape and convert into a shaded 3D polygonal representation of the terrain surface. The generated terrain surface is saved into a data base file for later editing.

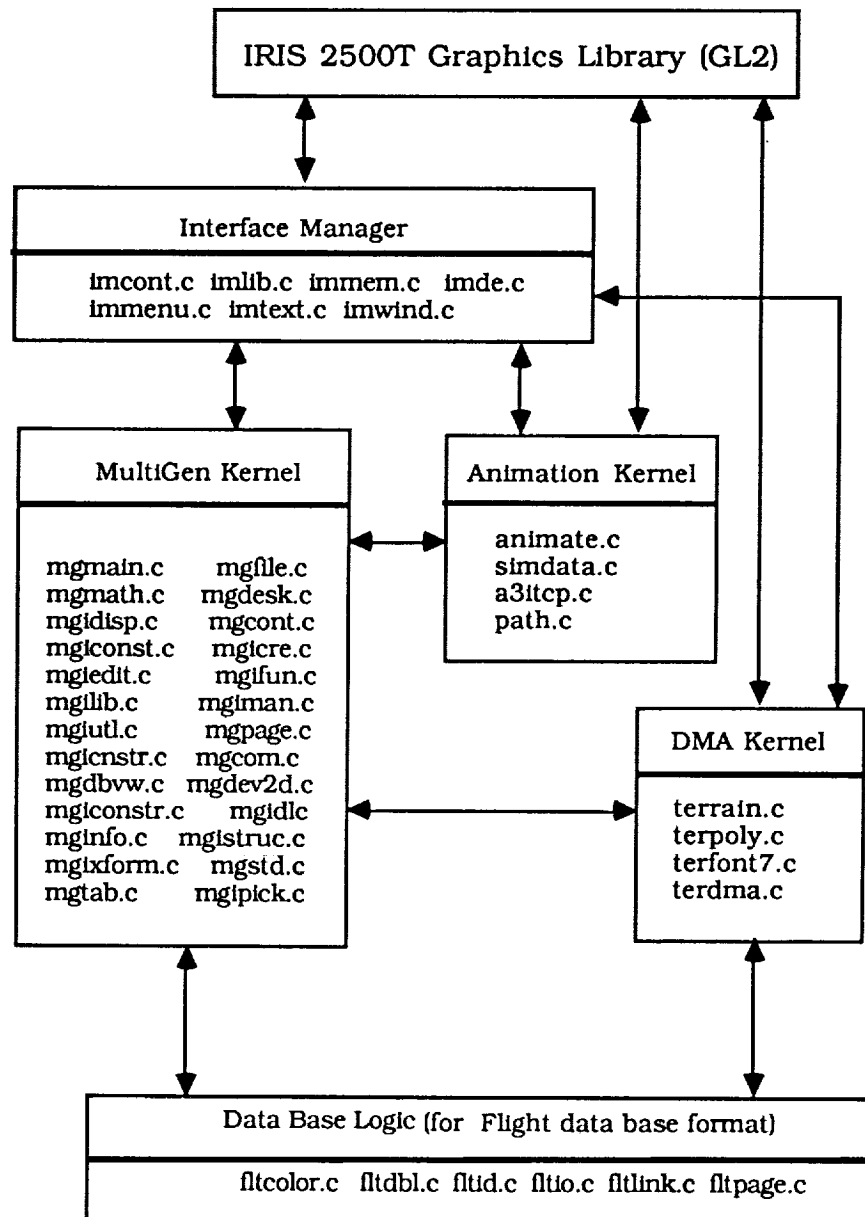


Figure 1 Organization of files in Views software

4.2 Unit Detailed Design

4.2.1 Interface Manager Subsystem (IM)

The Software Systems' Interface Manager Subsystem (IM) is a collection of C procedures that support a mouse and window oriented user interface on the Silicon Graphics Iris workstation. It provides the interface between the user and the application program. The Interface Manager imposes an architecture on the application program that is appropriate for a random sequence input from the user.

Basically, the Interface Manager (IM) source code is organized in seven separately compilable source modules. Externally callable procedures in the IM are preceded by a three letter module prefix and an underscore. Table 4-1 shows the modules that contain the source code for the IM.

<u>Module Name</u>	<u>Prefix</u>	<u>Purpose</u>
imwind.c	IWM_	Desktop and window managers
imlib.c	ILI_	Interface manager library
immem.c	IMM_	Memory management
imcont.c	ICM_	Control manager
imtext.c	ITE_	Text editor manager
immenu.c	IME_	Menu manager
imde.c	IDE_	Message and documenter/editor manager

Table 4-1. Interface Manager Source Modules

Detailed information on the Interface Manager Subsystem environment, procedures and organization can be found in *Software Systems MultiGen , Programmer's Guide to the Interface Manager*, Release 3.0, September, 1988. This documentation only describes those changes made by the A³I staff for our simulation. The source code of the entire Interface Manager Subsystem resides in "Orca", our (IRIS 2500T), under the directory of "/f1/mg3.0/Im".

4.2.1.1 The IM data structures: imstrucs.h

The definitions of IM data structures and defined constants are found in the file "imstrucs.h". A four-byte integer word (windowtype) has been added to the window data structure " windowstruc". The purpose of windowtype is used to specify the type of window for animation. If windowtype is zero, this window is a regular MultiGen window. If windowtype is 1, this window is for "World View" during the simulation. If windowtype is not equal to 1 or 0, this window is for observer's view. The following is a complete revised listing of the window data structure which is defined in file "imstrucs.h".

```
typedef struct wstruc {
    struct wstruc *nextwindow; /* pointer to next window below */
    struct wstruc *lastwindow; /* pointer to window above this one */
    int active;                /* true if this window is active */
    int shadow;                /* TRUE to draw a shadow around window */
    rectangle view;            /* the window in screen coords */
    point contentdelta; /* the content location from window lower left */
}
```

```

point contentsize; /* the size of content or offset from upper right */
rectangle content; /* the rectangle surrounding content */
int windowcontrols; /* the window controls */
int type; /* the window type, DESKTOPWINDOW,... */
int wdl; /* a pointer to the window's display list */
controlpt control; /* pointer to 1st control in window */
textedpt texted; /* pointer to 1st text in window */
char texton; /* TRUE if text edit active */
docedpt de; /* pointer to doced info if doced window */
int backgroundcolor; /* the color to paint window background */
char title [ MAXTITLE ]; /* the title of the window, NULL if none */
int misc; /* can be used for any purpose by user */
int ( *redraw ) (); /* the procedure to draw the contents */
int ( *contentproc ) (); /* called when mouse pressed in content */
int ( *deactiveproc ) (); /* called when window deactivated */
int ( *trackproc ) (); /* called when mouse over window and no
                        buttons down */
int ( *toppermproc ) (); /* called to ask permission to bring window
                        to top */
int ( *topproc ) (); /* called when window brought to top */
int windowtype; /****** A3I *****/
} windowstruc, *windowpt;

```

4.2.1.2 The Window Manager: imwind.c

The window manager is a set of functions that work with window data structures to manage and draw windows. Also, it has a set of functions to service events in the Iris event queue and mouse buttons. When user input is sensed, it called the appropriate IM or application procedure to service the event. Two functions, namely IWM_windowinit () and IWM_waitforevent (), in this file have been modified for the A³I simulation.

- IWM_windowinit () - A statement "IWM_communication = FALSE" is added at the end of this function which is used to set the current state to the regular "create and edit" mode, i.e. not in animation mode.
- IWM_waitforevent () - This procedure provides an infinite loop which polls the Iris event queue and mouse buttons. In addition to that, statements are added to check whether the Views software is in the animation mode. If it does, a call to the procedure ANA_communicate (), which will be described in detail in Animation Kernel Subsystem, is made in order to process this option.

4.2.2 MultiGen Kernel Subsystem (KER)

Detailed information on the MultiGen Kernel Subsystem environment, procedures and organization can be found in *Software Systems MultiGen , Programmer's Guide to the Interface Manager*, Release 3.0, September, 1988.

4.2.3 Data Base Logic Subsystem (DBL)

The purpose of the Data Base Logic Subsystem (DBL) is to provide data base independence in MultiGen; a different DBL is written for each data base format that operates with the Views software. Its functions are to access the data base file for MultiGen Kernel, to handle user interface issues such as coordinate display, terminology and modes, and to provide special purpose menu functions for the data base format.

A DBL subsystem is implemented as a series of low level procedures that are called from the MultiGen Kernel via the DBL linkage. They have a strictly defined function and calling sequence and are invoked to perform some task for the Kernel that can only be done by a procedure that understands the data base format and how it relates to the MultiGen internal format (ifmt). DBL is a subsystem, and as such it can have it own set of global and static variables low level utility procedures.

DBL procedures are called from the Kernel through a DBL linkage module. The use of this module allows all DBL procedure names to have the module prefix, so that a call to the DBL from the Kernel is always to a procedure with a DBL_ prefix. The DBL linkage module makes the procedure linking between the Kernel and the DBL very tight. The DBL linkage procedures should never contain programming logic. They call another procedure with the same suffix and calling conventions as the DBL procedure. A complete description of the Data Base Logic subsystem can be found in *Software Systems MultiGen, Programmer's Guide to Writing MultiGen DBL*, Release 3.0, September, 1988. The following is a description to those changes made by A³I staffs.

Phase III Views software adopted a simplified data base format called "Flight" (FLT) which is developed by Software Systems to support both simple and relatively sophisticated real time software applications. For a complete description on the Flight data base format, please consult *Data Format Specification for Software Systems Flight Data Bases*, Format Release 4, July 28, 1988. The source code of the Flight data base logic is organized in six separately compilable source modules and can be found in orca (Iris 2500T) under the directory of "/f1/mg3.0/Mg/Flt". Table 4-2 shows the modules that contain the source code for the DBL.

<u>Module Name</u>	<u>Prefix</u>	<u>Purpose</u>
--------------------	---------------	----------------

fltlink.c	DBL_	MultiGen to DBL linkage routines
fltcolor.c	FLC_	Flight data base color routines
fltdbl.c	FLT_	Flight data base logic routines
fltld.c	FID_	Flight DBL identification related processes
fltio.c	FLIO_	Handle input and output DBL for the Flight data base
fltpage.c	FPG_	Related to modify attributes command by accessing the data base attributes for ifmt beads

Table 4-2 Data Base Logic Source Modules

4.2.3.1 The Flight Data Base data structures: fltfmt.h

The definitions of the Flight data base data structures and defined constants are found in the file "fltfmt.h". A slight modification has been made to this file so that color can be assigned to the vertices of a polygon. The following is a revised listing for the vertex's data structure.

```

/* vertex record (no longer used: contains the id) */
typedef struct {
    rechdr rheader;
    icoord p; /* coordinate */
    /****** A3I *****/
    short vcolor; /* vertex color */
    /****** A3I *****/
} fvertex, *fvertexpt;

/* short vertex record */
typedef struct {
    srechdr rheader;
    icoord p; /* coordinate */
    /****** A3I *****/
    short vcolor; /* vertex color */
    short spare;
    /****** A3I *****/
} fvertexs, *fvertexspt;

```

4.2.3.2 The Flight Data Base color routines: fltcolor.c

The file "fltcolor.c" has a set of functions to define the color of the Flight data base format, to display the color palette and to modify the color definition. The DBL is assigned a subset of the workstation color space to map into data base format's color palette. Several functions in this file have been modified due to the additional color requirements of the A³I Cockpit Display Editor (CDE) CSCI. The name of these functions are FLC_colormap (), FLC_iris2dbl (), and FLC_dbl2iris ().

- FLC_colormap () - Originally, the starting color map index for the Flight data base format is specified by the MultiGen Kernel. However, due to the requirement for the CDE to replicate the wide range of colors found in a typical cockpit, a set of color indexes ranging from the starting color map index (specified by the MultiGen Kernel) to 511 is reserved. Therefore, the starting workstation color index number is changed to 512. It is accomplished by redefining a local global variable "Colorstart" to 512. The statement - "Colorstart = 512" is added before calling the procedure FLC_loadmap ().
- FLC_iris2dbl () - The data base color indexes for the Cockpit Display Editor are saved in terms of negative numbers. Therefore, a special provision is added to check the value of the Iris workstation color index. If the value of the workstation color index is less than "Colorstart", then this number is subtracted from "Colorstart" before returning to the calling function.
- FLC_dbl2iris () - The data base color indexes for the Cockpit Display Editor are saved in terms of negative numbers. Therefore, a special provision is added to check the sign of the data base color index first before converting to Iris workstation color index. If the data base color index is negative, then a value equivalent to "Colorstart" is added to this data base color index before returning to the calling function.

4.2.3.3 The Flight Data Base logic routines: fltdbl.c

The default unit of measure for the Flight data base is changed from meters to feet. It is accomplished by commenting the original statements and replaced by the following statements.

```
/* Original statements */
/*char Unitstring [ 6 ] = "m"; /* current data base unit string */
/*int Flt_units = 0; /* current data base unit code */
/*int Flt_udiv = -100; /* current data base divisor */
/*char Lfmt [ 40 ] = "%11.2f%s\n%11.2f%s\n%11.2f%s\n"; */
/*char Sfmt [ 10 ] = " %4.2f%s"; */
/* New statements */
char Unitstring [ 6 ] = ""; /* current data base unit string in feet */
int Flt_units = 4; /* current data base unit code in feet */
int Flt_udiv = -3072; /* current data base divisor in feet */
char Lfmt [ 40 ] = "%11.4f%s\n%11.4f%s\n%11.4f%s\n";
char Sfmt [ 10 ] = " %6.4f%s";
```

- FLT_init () - The auto-write option is deleted by commenting out the statement which calls the function IME_checkcommand ().
- Grid_init () - The default grid spacing is redefined according to the selected internal resolution "COM_internal_resol".
- Si2float () - A check to the variable "DH_Displaycoordtype" is added. If "DH_Displaycoordtype" is equal to 2, then outputs the results in terms of inches in the tracking window.
- Float2si () - A check to the variable "DH_Displaycoordtype" is added. If "DH_Displaycoordtype" is equal to 2, then accepts the user input in terms of inches from the tracking window.
- FLT_dmacolor () - The number of colors for DMA option is changed from 11 to 74. This change adds more colors to the generated terrain surface.

4.2.3.4 The Flight Data Base input and output routines: fltio.c

The file "fltio.c" has a set of procedures to handle the input and output functions for the Flight data base. These functions include the opening and closing the data base file, reading the information from the data base file and putting these information into the MultiGen kernel's ifmt, and acting as data base logic menu handler. Since the Flight data base format is still in evolving stage, therefore, modifications are required in order to read the earlier versions of Flight data base files. A local static variable "Oldfile" is added to this module to identify whether the incoming file is in the latest version of Flight data base format or not.

- Builddbl () - A provision is added to check the file header. If the variable "ihattr->attributes.udiv" is equal to zero, then set the variable "Oldfile" to TRUE. Consequently, the variables "ihattr->attributes.units" and "ihattr->attributes.udiv" need to be reset to 4 and "-COM_internal_resol" respectively.
- *F2ibead () - All modifications made in this procedure are related to adding an extra argument in function IL_addivertex (). Please refer to module mgilib.c for more detail.

4.2.3.5 The linkage between the MultiGen Kernel and the Flight Data Base logic routines: fltlink.c

All modifications made in this module are related to controlling the display of either the data base attribute window or the descriptive data base window. The variable "DH_Displaycdetype" defines the type of display which is specified by the user. A provision to check the value of the variable "DH_Displaycdetype" is added to the following routines - DBL_mod_attr (), DBL_getfieldtype (), DBL_getelement (), DBL_getfield (), DBL_getstring (), DBL_putfield (), DBL_pagegeneral ().

4.2.4 Animation Kernel Subsystem (ANA)

The Animation Kernel subsystem is an option developed by A³I staff as an added on to MultiGen system. Its main purpose is to generate different views for displaying the events of the A³I mission simulation in 3D graphics form. Windows are displayed on the screen to present the world view, observer's view, lead or wing helicopter view and etc. The user has the option to control the number of windows to be presented. The Animation kernel subsystem obtains simulation data, such as the location of the world objects, from a predefined standalone data file or from the simulation executive program through inter-process communication pipeline.

The source code of the Animation Kernel is organized in three separately compilable source modules. Externally callable procedures in this subsystem are preceded by a three letter module prefix and an underscore. Table 4-3 shows the modules that contain the source code for the Animation kernel. The source code of the following files can be found in orca (Iris 2500T) under the directory of "/f1/mg3.0/a3i". Figure 2 shows a layout of the Animation subsystem kernel in block diagram form.

<u>Module Name</u>	<u>Prefix</u>	<u>Purpose</u>
--------------------	---------------	----------------

animate.c	ANA_	Animation control module
path.c	PAT_	Flight path selection module
simdata.c	SIM_	Reading simulation data module

Table 4-3. Animation Kernel Source Modules

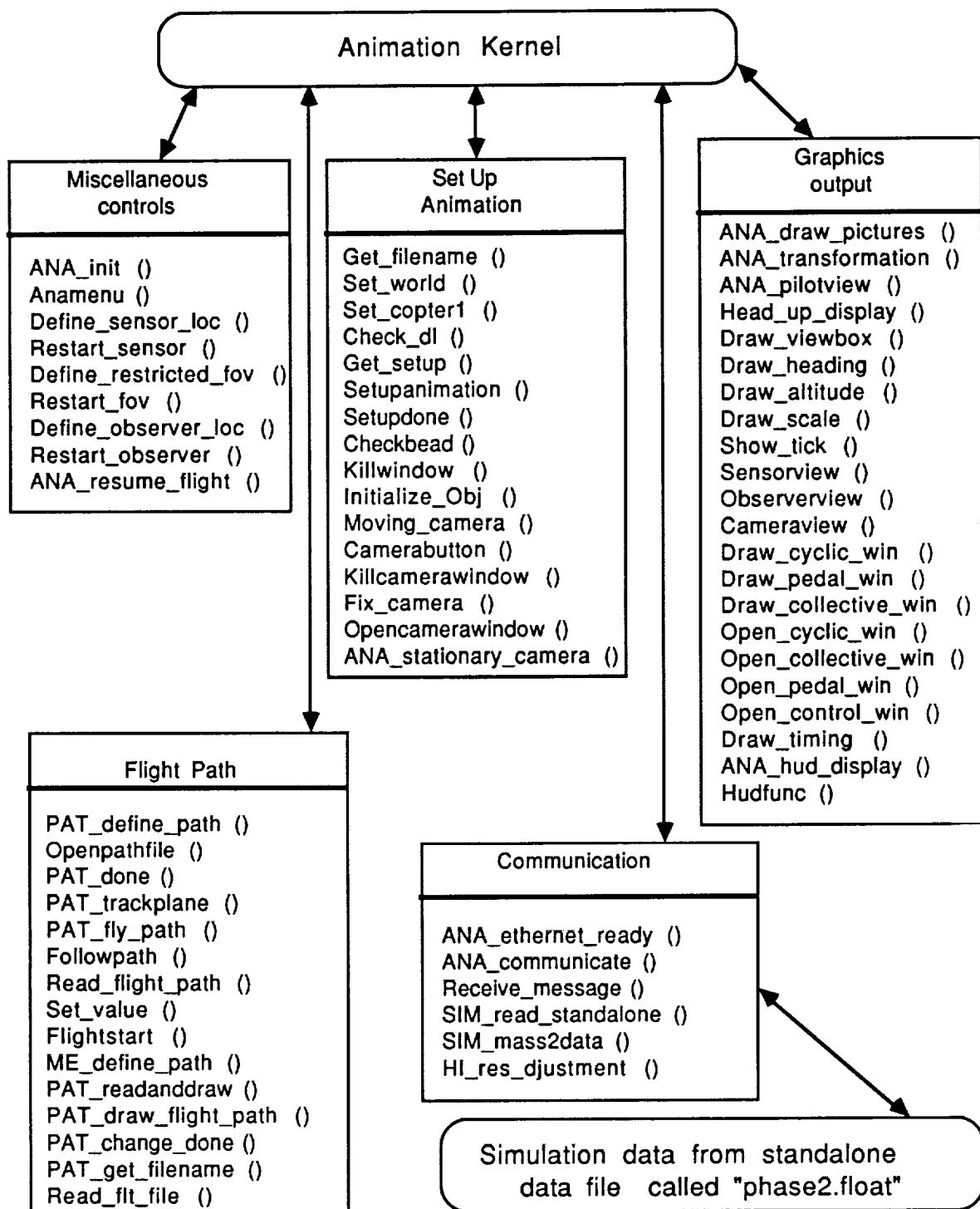


Figure 2 Layout of Animation Subsystem Kernel

4.2.4.1 The Animation control module: animate.c

4.2.4.1.1 Initialize Animation menu to the MultiGen system: ANA_init ()

The procedure ANA_init () is called by the main procedure in MultiGen Kernel subsystem. The primary purpose of this procedure is to add the animation menu to the MultiGen's pulldown menu system and initialize the animation message file. Two files are opened by ANA_init ():

animate.menu - a text file for animation pull-down menu.

animate.msg - a text file for defining the messages and templates used by animation kernel.

Please consult *Software Systems MultiGen, Programmer's Guide to the Interface Manager*, Release 3.0, September, 1988. for detailed information on how to set up the menu and message files.

4.2.4.1.2 Control of ANA pulldown menu selection: Anamenu ()

The function Anamenu () is called when user places the cursor on the Animation menu and picks an item from the pulldown menu.

```
Anamenu ( m, id )    /* called when ANA pulldown menu selected */
menupt m;           /* animation menu pointer */
int id;             /* the number of the selected item */
```

The purpose of this function is to accept the user's input, i.e., the item selected from the menu, and to call the appropriate procedure to process the command. The following is a listing of all the commands which can be found in the animation menu:

1. Setup - to get user input for setting up the animation process.
2. World View - to open up a window for showing the world view.
3. Moving camera - to open up windows for showing pilot/driver view.
4. Fix camera - to define and open up windows for observer's view.
5. Head Up Display - to turn on/off the head up display mode.
6. Ethernet - to start the animation process.
7. Stop Animation - to stop the animation process.
8. Resume Animation - to resume the animation process.
9. Define Flight Path - to define a flight path for the lead helicopter.
10. Fly Thru Path - to run the animation using a predefined flight path file.
11. Freeze - to temporary stop the animation.
12. Single Tick - to run the animation in a tick by tick base.
13. Restricted Pilot view - to show the pilot view in a restricted pilot view form.
14. Define Field of View - to define the field of view for the restricted pilot view.
15. Define Sensor Location - to specify the location of a sensor for sensor's view.
16. Full Screen - to display the current window using the whole screen.
17. 4D to Iris - to send the simulation data from one Iris machine to another Iris machine (from coral to manta only).
18. Show Timing - to show the time it takes to render one tick of the simulation picture.
19. Control Display - to display the cyclic, collective and pedal controls in symbolic form.
20. Define Observer's Location - to define the location of the observer relative to the lead helicopter.

21. Draw Flight Path - to draw the user's defined flight path.
22. Change Way Point - to change the flight path while in the middle of executing the "Fly Thru Path" command.
23. HUD - to turn on/off the HUD.

4.2.4.1.3 Get the animation set up file name: Get_filename ()

The purpose of this procedure is to let the user type in the name of the set up file. It is called when the user selects the "Setup" command from the Animation menu. The default set up file name is "768animateflt.doc", however, the user can redefine it by typing in another file name. Once the user hits the carriage return key, function Get_setup () is called.

4.2.4.1.4 Read the content of the set up file: Get_setup ()

The purpose of function Get_setup () is to open the set up file and read its content. The content of the file is arranged so that it contains the name of the world objects and their corresponding graphics identification number. In addition, it contains the graphics identification numbers of the cockpit instruments for the lead helicopter. Each line in this file represents one moving object in the simulation world. The first entry of every line is the object's name. The second entry is the graphics identification number of the object used in MultiGen. The third entry is the graphics identification number of the same object after destroyed or exploded. The sequence of these objects in this file dictates the order of displaying these objects during animation. If cockpit instruments are included in the simulation, enter an asterisk on a new line and then follow with their graphics identification numbers. A sample animation set up file is listed in the following figure.

Lead_Helicopter_H419	LEAD	0	
Wing_Helicopter_H836	WING		0
Jeep	JEEP	0	
Missile_Launch_1	LAUNC	0	
Tank	TANK		0
Truck_1	TRUCK	0	
*			
G1	G2	G3	G4
G5	G6	G7	G8
G9	G10	G11	G12
G13	G14	G15	G16
G17	G18	G19	G20
G21	G22	G23	G24
G25	G26	G27	G28
G29	G30		

Figure 3 The content of animation set up file

After it finishes reading the whole set up file, function Setupanimate () is called to echo the information.

4.2.4.1.5 Echo the content of the set up file: Setupanimate ()

Function Setupanimate () is used to display the content of the set up file and the user can interactively change the information if desired. This is limited to echoing the information of the world objects only and no information on the cockpit instruments will be shown. Function Setupdone () will be called when the user is satisfied with all the input and hits the "DONE" button in the echo window.

4.2.4.1.6 Check the validity of the world objects: Checkbead ()

All user specified world objects in the set up file will be checked for their existence in the data base file by function Checkbead () which is called by function Setupdone ().

```
ibeadpt Checkbead ( w, string )
windowpt w; /* current data base window pointer */
char *string; /* graphics id string of the world object */
```

The ibead pointer is returned to the calling routine if the id string of the world object specified by the user is found in the data base file. Otherwise, a null pointer is returned.

4.2.4.1.7 Finish the set up process: Setupdone ()

This function is called when the user hits the "DONE" button in echo window. It calls function Checkbead () to check the validity of all user specified world objects and then updates the value for variables "Max_cockpit_items" and "Max_object". Next, it calls function Set_world () to display the world view window. In addition to the above functions, this function also performs the preparation works for moving camera view and stationary camera view. In order to display those view, a rotation matrix (Ground_matrix) is needed. It can be obtained by rotating the Iris' ground level plane (XZ plane) to A³I model's ground level plane (XY plane). The following code segment illustrates how it works :

```
pushmatrix (); /* save the original matrix in matrix stack */
loadmatrix ( Identity ); /* put an identity matrix at the top of the matrix stack */
/* Iris Z-axis is pointing outward and X-axis is pointing to the right */
rotate ( -900, 'x' ); /* bring Z-axis pointing upward */
rotate ( 900, 'z' ); /* bring X-axis pointing inward */
getmatrix ( Ground_matrix ); /* save the matrix in Ground_matrix */
popmatrix (); /* restore the original matrix from stack */
```

For stationary camera's views, the stationary window array is initialized. Each element of the stationary window array "Camera_posptr []" is a record. The type definition of the record is as following:

```
typedef struct { /* record for all stationary camera */
    int id; /* window identification number */
    float x, y, z; /* location of the stationary camera */
    int yaw; /* look at direction of the stationary camera */
} fixcamera, *fixcameraptr;

static fixcameraptr Camera_posptr [ 10 ];
```

4.2.4.1.8 Open the world view window: Set_world ()

Function Set_world () is called either by function Setupdone () or when the user selects the World View command from the Animation menu. The main purpose is to open up a window to show the god's eye view of the gaming area and redefine some of the viewing parameters.

4.2.4.1.9 Open the moving view window: Set_copter1 ()

Function Set_copter1 () is called when the user selects the "Moving Camera" command from the animation menu. The main purpose of this function is to open up a window to show what the pilot/driver would see inside the vehicle.

4.2.4.1.10 Do transformation on object for world view: ANA_transformation ()

Procedure ANA_transformation () is called by function Drawbead () in file module mgidisp.c. The following statements define the header of this procedure.

```
ANA_transformation ( ib )
ibeadpt ib;      /* world object's ibead pointer */
```

The main purpose of this procedure is to move the object to its new location in the simulation world when the world view is displayed. It is accomplished by doing a translation to the new position and then the rotations about the yaw, pitch, and roll axes. Due to different definition of the north direction in the graphics world and the object world, a subtraction of 90 degrees from yaw is required. The pointer for the object's dynamic values is obtained from the incoming ibead pointer "ib->gdbinfo".

4.2.4.1.11 Do transformation on the object for pilot view: ANA_pilotview ()

Procedure ANA_pilotview () is called by function IDR_setport () in file module mgidisp.c. The following statements define the header of this procedure.

```
ANA_pilotview ( w, pitch, roll, yaw, tranx, trany, tranz )
windowpt w; /* window pointer for the current window */
float pitch, roll, yaw;
float tranx, trany, tranz; /* not used */
```

The main purpose of this procedure is to form a transformation matrix when either the moving camera view or the stationary camera view is displayed. This procedure checks the value of the variable "windowtype" to determine the calling procedure. If the current window is for a stationary camera view, procedure Cameraview () is called. If the current window is for the lead helicopter observer's view, procedure Observerview () is called. If the current window is for either the lead or wing helicopter's sensor view, then procedure Sensorview () is called. If none of the above conditions are met, the current window is for the pilot's view which is attached to one of the world objects. With the vehicle's new position and rotation, a transformation matrix is formed first by rotating about the roll, pitch, and yaw axes respectively. Next it is translated to its new position, and finally, multiplied by the ground matrix which was formed in function Setupdone (). This newly formed matrix is saved on the matrix stack and is used when the view is drawn.

4.2.4.1.12 Form transformation matrix for sensor's view: Sensorview ()

Procedure Sensorview () is called by function ANA_pilotview. The sensor is attached to either the lead or wing helicopter. This procedure sets up a transformation matrix on top of the graphics matrix stack for displaying the sensor view. This is accomplished by performing successive coordinate transformations. First of all, the location of the sensor is moved from the origin of the helicopter to the user defined location. Next, it applies rotation successively about roll, pitch and yaw following by translation to the helicopter's new position. Finally, the current matrix is multiplied by the ground matrix. The newly formed matrix is saved on the graphics matrix stack and is used when the sensor view is drawn.

4.2.4.1.13 Form transformation matrix for sensor's view: Observerview ()

Procedure Observerview () is called by function ANA_pilotview. An observer's view is used to monitor the lead helicopter's movement only. This procedure sets up a transformation matrix on top of the graphics matrix stack for displaying the observer's view. The algorithm to obtain the transformation matrix is very similar to procedure Sensorview (). First of all, the location of the observer is moved from the origin of the helicopter to the observer's location. Next, it applies rotation about the yaw axis and follows with a translation to the lead helicopter's new position. Finally, the current matrix is multiplied by the ground matrix. The newly formed matrix is saved on the graphics matrix stack and is used when the observer's view is drawn.

4.2.4.1.14 Form transformation matrix for fixed camera view: Cameraview ()

Procedure Cameraview () is called by function ANA_pilotview. A fixed camera view is used to monitor the activities inside the gaming area. This procedure sets up a transformation matrix on top of the graphics matrix stack for displaying the fixed camera view. The algorithm to obtain the transformation matrix is as following. First, it applies rotation about pitch and yaw axes, followed by a translation to the fixed camera location. Finally, the current matrix is multiplied by the ground matrix. The newly formed matrix is saved on the graphics matrix stack and is used when the fixed camera view is drawn.

4.2.4.1.15 Prepare for animation: ANA_ethernet_ready ()

This procedure is called when the user selects the "Ethernet" command from the animation menu. The purpose of this procedure is to set the Views software in animation mode. First of all, it checks whether the set up procedure has been done or not. If no set up procedure is done before, it signals the user and returns to MultiGen's editing mode. Secondly, it sets up the memory space for each world object for storing the animation data by calling procedure Initialize_obj (). Next, it checks whether standalone mode or communication mode is used. If standalone mode is used, it opens the data file called "phase2.float" which can be found in the current directory. Finally, the Views software is turned into animation mode by setting the global variable "TWM_communication" to true.

4.2.4.1.16 Ready to read animation data: ANA_communicate ()

When the global variable "TWM_communication" is set to true, then this procedure is called by function IWM_waitforevent () inside the event loop. The purpose of this procedure is directed the Views software to get the simulation data from the right place. If the animation is in standalone mode, it calls procedure SIM_read_standalone (), otherwise it calls procedure Receive_message (). Finally, it calls ANA_draw_pictures () to render the simulation on the screen.

4.2.4.1.17 Draw all windows: ANA_draw_pictures ()

Once the Views software receives the simulation data for the current tick, procedure ANA_draw_pictures () is called to draw the scene of the simulation for all active windows.

4.2.4.2 The Flight path module: path.c**4.2.4.2.1 Define the flight path: PAT_define_path ()**

This procedure is called by function Anamenu () when the user selects the "Define Flight Path" command from the animation menu. It calls function Openpathfile () which prompts the user to type in a file name for saving the flight path information. Next, it calls function ME_define_path () to set up a menu for the user to pick the way points of the flight path.

4.2.4.2.2 Save user defined flight path: PAT_done ()

Once the user finishes defining the flight path and picks the "DONE" button in the editing window, procedure PAT_done () is called. The purpose of this procedure is to terminate the inputting process and save all input data into a ASCII file whose name is specified by the user. Each line in this file stores the information for one way point which consists of station number, X, Y, and Z locations, altitude, air speed, and heading.

4.2.4.2.3 Locate the terrain surface: PAT_trackplane ()

This function is called by function MC_mouse2vtx () in file module mgiconst.c. The main purpose of this function is to find the plane equation of a terrain surface when the user places the cursor on top of a terrain surface and clicks the mouse button. The elevation at that particular location can be calculated with the provided plane equation.

4.2.4.2.4 Fly Through the flight path: PAT_fly_path ()

This procedure is called when the user selects the "Fly Thru Path" command. It prompts for the flight path data file name, the flying distance between points, and the elevation's scaling factor. Next, it calls function Followpath () to verify the user's input and then calls function Flightstart () to start the flying through command.

4.2.4.2.5 Draw the flight path: PAT_draw_flight_path ()

This procedure is called when the user selects the "Draw Flight Path" command. It checks whether a flight path has been drawn before, if it has, the existing flight path is deleted from the display list and a new flight path is redrawn on the screen.

4.2.4.3 Read simulation data module: simdata.c

4.2.4.3.1 Read data from a standalone file: SIM_read_standalone ()

This procedure is called only when the simulation is in standalone mode. The purpose of this procedure is to read the simulation messages from the data file called "phase2.float" and assign those messages to world object's data structure. The content of a complete data structure of the world objects is listed in Appendix. This procedure also assigns lead helicopter's dynamic values to the cockpit instruments if they are defined by the user.

4.2.4.3.2 Read data from pipe file: SIM_mess2data ()

This procedure is called only when the simulation is in integrated communication mode. The purpose of this procedure is very similar to procedure SIM_read_standalone () except it reads in simulation messages from a pipe file called "a3i1000".

4.2.4.3.3 Terrain location adjustment: Hi_res_djustment ()

The Iris Graphics Engine is limited to accepting an integer value no larger than 24-bit. The physical location of the A³I gaming area is 10 miles away from the graphics origin, therefore, after converting the coordinates of the terrain surfaces into the internal format of MultiGen in high resolution mode, most of the values are bigger than 24-bits. In order to circumvent this problem, the origin of the gaming area in the graphics world is moved from (64741.0, 65620.0, 0.0) to (0.0, 0.0, 0.0) and procedure Hi_res_djustment () is used to make an appropriate adjustment after checking the value of COM_internal_resol.

```
Hi_res_djustment ( zerox, zeroz )
float *zerox, zeroz;          /* adjusting values in x and y directions */
```

If the value of COM_internal_resol is greater than 32, which is the normal internal resolution for MultiGen, then the returned values are 64741.0 and 65620.0 for x and y directions respectively. Otherwise, zeros are returned.

4.2.5 DMA Kernel Subsystem (TER)

The DMA Kernel subsystem is an option developed by the A³I staff as an add-on to MultiGen and later on modified by Software Systems to make this subsystem more user friendly. The main purpose of this option is to extract elevation data from standard level 1 DMA DTED distribution tapes and convert into polygonal surfaces. The generated terrain surface is saved into a data base file for later editing.

The DMA Kernel subsystem is divided into two parts -

- (i) A standalone preprocessor is used to read elevation data off the distribution tape and save into a data file called post file. The name of this program is called "readdma" which is resided in the directory called "/f1/mg3.0/Mg/Dma" in orca (Iris 2500T).
- (ii) The second part is built into the MultiGen system. The post file saved in the first part becomes the input file for this part which converts the elevation data into polygonal surfaces. The source code of this part is organized in three separately compilable modules. Externally callable procedures in this subsystem are preceded by a three letter module prefix and an underscore. Table 4-4 shows the modules that contain the source code for the DMA kernel. The source code of the following file can be found in orca (Iris 2500T) under the directory of "/f1/mg3.0/Mg/Dma".

Module Name Prefix Purpose

terrain.c	TER_	DMA user interface module
terdma.c	TP_	Read post file module
terpoly.c	TPOL_	Generate terrain polygons module

Table 4-4. DMA Kernel Source Modules

4.2.5.1 The standalone preprocessor module: readdma.c

This program is used to extract elevation data from DMA DTED distribution tapes or a tape image disk file. A tape image disk file is a file which has identical format with a given distribution tape. The detailed format of the distribution tape is described in *Defense Mapping Agency (DMA), Product Specifications for Digital Terrain Elevation Data (DTED), PS/ICD/200, PS/ICF/200,*

Second Edition, St. Louis, Missouri, April, 1986. The main goal of this program is to follow the format of the tape and extract the required data.

The main program controls the flow of the whole program. It checks the command line input to see whether the input is coming from a distribution tape or a tape image file. If the second argument in the command line is "file", then the file name of the DMA's DTED tape image will be asked. Otherwise, it is assumed that a distribution tape is already mounted on the tape drive. Figure 4 shows a flow chart of this program. Additional information on each routine can be found in the documentation prepared by Software Systems. A post file will be saved in the disk when the program is finished. This post file becomes the input file to the second part of the DTED conversion.

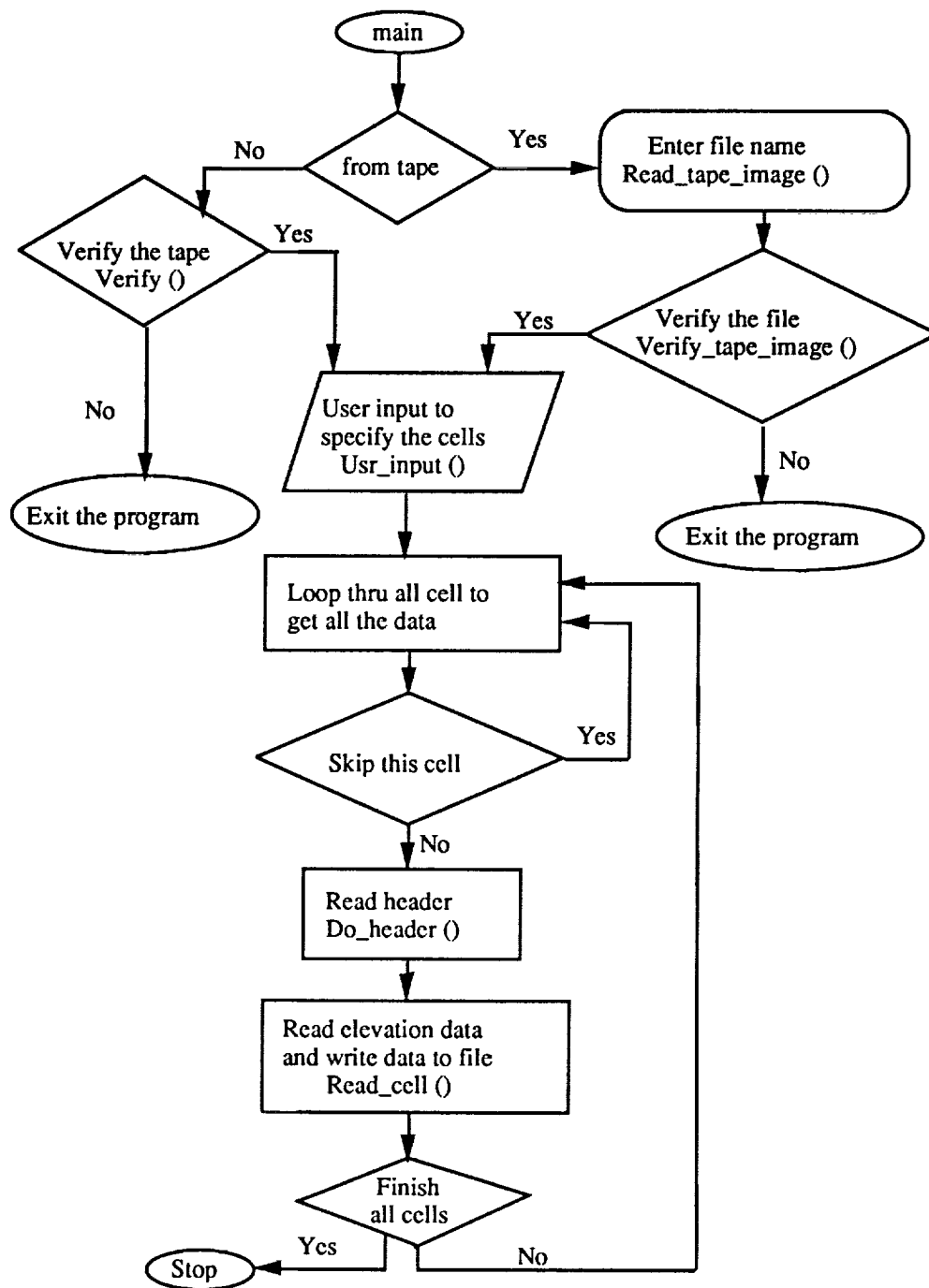


Figure 4 A flow chart for program readdma.c

4.2.5.2 Converting elevation data into polygonal surfaces

4.2.5.2.1 Read post file module: terdma.c

The procedure TP_getfile () in this module is used to check and read user specified post file. The format of a post file is described as follows:

Record type 1 : size = 128 bytes
Contents : file header block which is defined by the following structure -

```
typedef struct ss_stdhdr {      /* standard file header block (128) */
    long len;                   /* size of header block in byte */
    char id [ 4 ];              /* file identification code "SSYS" */
    char part [ 4 ];            /* software part no. */
    char rev [ 8 ];             /* software revision no. */
    char create [ 26 ];         /* creation date/time */
    char update [ 26 ];         /* last update date/time */
    char spare [ 52 ];          /* spare */
} ss_stdhdr, *ss_stdhdrpt;
```

Record type 2 : size = 4 bytes (binary)
Contents : total number of cells in this file. (floating point)

Record type 3 : size = 2 bytes (binary)
Contents : minimum elevation in meters. (integer)

Record type 4: size = 2 bytes (binary)
Contents : maximum elevation in meters. (integer)

Record type 5 : size = 40 bytes (binary)
Contents : minimum latitude value. (floating point)
 maximum latitude value. (floating point)
 minimum longitude value. (floating point)
 maximum longitude value. (floating point)
 latitude data interval in second. (floating point)
 longitude data interval in second. (floating point)
 number of latitude points. (floating point)
 number of longitude points. (floating point)
 distance between longitude points in feet. (floating point)
 distance between latitude points in feet. (floating point)

Record type 6 : size = number of latitude points * 2 bytes (binary)
Contents : elevation data in meters. (integer)
 Repeat this record for X times where X is equal to number of longitude
 points in one cell.

Note: Repeat record types 5 and 6 for N times where N is equal to total number of cells in this post file.

Once the user selects a portion of area from the origin post file and desires to save it, procedure TP_newfile () is called to save the selected area into a post file format. Saving terrain post data for

smaller areas of interest within the original gaming area guarantees that the area of work will be exactly the same in subsequent modeling sessions.

4.2.5.2.2 DMA user interface module: terrain.c

The procedures in this file are used to handle user interaction. It includes putting the DMA pull-down menu into MultiGen's pull-down menu system by procedure `TER_init ()`; opening post files by procedure `Open_terfile ()`; drawing grid maps and color filled contour maps on the screen by procedures `Draw_grid ()`, `Redraw_gridw ()`, `Do_contours ()`, and `Contours ()`; putting up control windows for the user to redefine the parameters for data conversion by procedure `Control_window ()`; and directing user selection to the appropriate routine by procedure `Termenu ()`. The detailed internal structure of this module can be found in the documentation prepared by Software Systems.

4.2.5.2.3 Generate terrain polygons module: terpoly.c

The procedures in this file are used to generate the polygonal surfaces from the elevation data. Procedure `TPOL_form_terrain_poly ()` is the control routine for forming the terrain surfaces. It is called when the user selects the "Polygons" command in the grid window. The whole terrain surface is divided into several groups of objects. Polygonal surfaces are formed by comparing the maximum elevation among four corner posts against the maximum and minimum elevation among all interior posts. If the difference between these two elevations is within a user defined tolerance, a single polygonal surface is formed providing that all four corners are on a same plane, otherwise 2 triangular surfaces are formed. If the difference is larger the tolerance, four triangular surfaces are formed by using four corner posts and the interior post of either maximum or minimum elevation. Procedures `Subdivide ()`, `Form_face_mat ()`, `Fill_array ()`, and `Form_face ()` are used to form the surfaces and to insert into MultiGen internal data format (ifmt) for display. The color of the polygonal surface is assigned according to the maximum elevation within the surface and color is assigned to the vertex according to its elevation. The detailed internal structure of this module can be found in the documentation prepared by Software Systems.

5.0 NOTES

5.1 Miscellaneous

5.2 Limitations

The limitations for using the Animation option are:

- i. All world objects have to be in a single FLIGHT database file. Each world object should be defined by only one ID in the Group level. Therefore, an object can be created individually and then this model is copied into the animation database file using the Group mode.
- ii. When the Cockpit Display Editor is included, only the instruments and gauges on the lead helicopter's cockpit panels can be animated. There is no provisions at this time to handle the animation of cockpit instruments for wing helicopter.
- iii. The maximum number of world objects can be handled at one time is 150 which includes the lead helicopter's cockpit instruments.

5.3 Future Directions

Integrating Views software into MultiGen provides a good set of tools for the user/designer to create geometric models of world objects and observe the simulation in different viewing perspectives. Basically, MultiGen is a modeling system for creating and editing of three dimensional graphics data bases, while Views provides tools for running and monitoring a simulation. Therefore, they are totally different in nature. After integrating them together, there is no performance changes in the editing part. However, the performance of the animation part is affected due to extra checkings in drawing mode which is designed for the editing part. In order to speed up the rendering time for each frame during simulation, the animation part should have its own drawing routines to by-pass all those unnecessary checkings.

Secondly, there are still no tools to extract cultural information from DMA's Digital Feature Analysis Data (DFAD) and overlay them on the DTED. Without any cultural information, important aspects of the world environment cannot be displayed.

6.0 USERS GUIDE

6.1 Introduction

This section describes how to use Views software/MultiGen and its installation procedures.

6.2 Related Documentation

Software Systems MultiGen - Modeller's Guide, Release 3.0, Software Systems, San Jose, September 1988.

6.3 Overview of Purpose and Functions

The Views software/MultiGen is a system for creating and editing three dimensional graphics data bases and providing tools for viewing the simulation sequence. The whole system is mouse oriented and visually intuitive. The program is controlled by pointing at symbolic graphics and menus on the graphics workstation display. This type of user interface was originally developed by The Xerox Palo Alto Research Center (PARC) and has been popularized by the Apple Macintosh computer.

6.4 Installation

6.4.1 Installing MultiGen Software Environment

The following procedure shows how to install the MultiGen Software environment:

- i. Put MultiGen release tape in the tape drive.
- ii. Change directory to the destination directory that will be the root directory of the entire MultiGen environment. In our case, directory "/f1/mg3.0" is the root directory for MultiGen in orca.

- iii. Type "tar xvo" and wait for down loading of the files. The correct directory structure will be created. Figure 5 shows the file structure for development of MultiGen/Views.

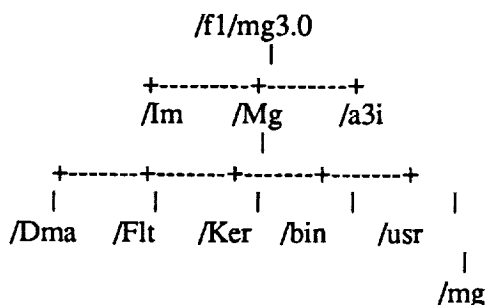


Figure 5 File structure for MultiGen

"Im" contains the files that make up the Interface Manager.

"a3i" contains the files that make up the Animation and CDE options.

"Dma" contains the files that make up the DMA terrain option.

"Ker" contains files that belong to the MultiGen Kernel.

All of above directories contain Makefiles to compile the source files, source files, include files and binaries. They also contain miscellaneous run-time menu and message resource files.

"Flt" contains the files that make up the Flight Data Base Logic and the main Makefile to compile and create the MultiGen/Views executable file. The content of this Makefile is showed in the Appendix A.

"bin" contains shell scripts needed for installing MultiGen, miscellaneous utilities, plus all ".tip" help files. It is recommended to put all MultiGen maintenance related utilities in this directory and set the path to it.

"mg" contains the run-time menu and message resource files, and the MultiGen executable file.

6.5 Start-up and Termination

All the graphical output from MultiGen/Views software are displayed on the system console of Iris 2500T. All FLIGHT database files are created by using MultiGen. The following instructions are used to create a new FLIGHT database file -

- i. Type "cd /f1/mg3.0/Mg/usr/mg", then press return key. It will bring to the directory where the MultiGen executable file is located.
- ii. Type "mgflt.good", then press return key which tells the operating to execute the MultiGen.

- iii. As soon as the desk top of MultiGen is set up, then the user has to choose the "new file" option under the "I/O" pull-down menu. From here on, the user can follow the MultiGen Modeller's Guide to create the model. Once the model is created, then the user has to select the "write file" option under the "I/O" pull-down menu to save the model into a disk file for future editing. To leave the program, drag the cursor down to "quit", then move the cursor to the right and select "confirm" under I/O menu.

The following instructions are used to execute MultiGen with an existing FLIGHT database file -

- i. Type "cd /f1/mg3.0/Mg/usr/mg", then press return key which will bring to the directory where the MultiGen executable file is located.
- ii. Type "mgflt.good filename", then press return which tells the operating to execute the MultiGen with the given file name called "filename".

6.6 Functions and Their operation

6.6.1 Setting up for Phase III demonstration

The Views software is executed on the system console of Iris 2500T. This is a standalone demonstration, i.e., it gets the simulation data from a data file called "phase2.float" which is resided in the same directory as the binary code of the View software. The following is a summary of step by step procedures for setting up the Views software for A³I demonstration.

Part 1 - The A³I World - Setup Procedures:

1. Login orca (Iris 2500T) as usual.
2. Type "cd /f1/mg3.0/Mg/usr/mg", then press return.
3. Type "mgflt.good 768/ah64.flt", then press return.
4. After MultiGen is running, re-size the database window to about 4" square and relocate it to the upper right corner of the screen. Pull down the "ANIMATION" menu bar and select the "Set Up" command.
5. Type in the "Set Up" filename which is called "768ani_flt3.doc" and press return. Next, a window will appear on the screen displaying the name and graphics ID of each object, pick the "DONE" button to terminate this input sequence if all of them are correct, otherwise, make the necessary changes before picking the "DONE" button.
6. A "World View" window is opened up at top right corner of the screen. Pull down the shade to display all control dials and then turn on the orthogonal display option. The user may need to resize the "World View" window in order to turn on the orthogonal display option.
7. Pull down the "ANIMATION" menu and select the "Fix Camera" command to define the observer's view inside the gaming area.
8. The fix camera location is specified by defining the camera location and its initial line of sight direction. Once those two information are defined, a camera window is popped up to display the view. Relocate the camera window to the upper left corner of the screen.
9. Relocate the "World View" window to the lower left corner and make certain that windows are not overlapped.
10. During demos, select "Ethernet", which can be found in the "ANIMATION" menu, to start the simulation.

Part 2 - Dynamics Analysis - Setup Procedures:

1. Once the Part I is over, move the "World View" window to upper right corner and cover up the database window.
2. Pull down the "ANIMATION" menu and select the "Moving Camera" command.
3. Select the "Wing Helicopter View" from the pop up menu. Next, a window for wing helicopter view is opened up on the bottom right corner of the screen.
4. Repeat Step 2 and 3, instead of selecting "Wing Helicopter View", select "Lead Helicopter View". Then the window for displaying lead helicopter view is opened up on the bottom left corner of the screen.
5. Pull down the "CDE" menu and select "Load Link File" command, and then type in the link file name called "radar_alt++.lnk".
6. Again, pull down the "CDE" menu and select "animate init" command to initialize the cockpit instruments. As soon as this command is done, press "m" two times for displaying the full detail of the whole cockpit.
7. Select "Ethernet", which can be found in the "ANIMATION" menu, to start the simulation again. The console shows the whole simulation in four different viewing windows at the same time - world view window, observer's view window, lead helicopter's pilot view, and wing helicopter's view.
8. As an option, similar set up can be done in manta (Iris 3120) to show the lead helicopter's pilot view using the whole screen.
 - a. Login manta as usual.
 - b. Type "cd /c1/tmh/mg3.0/Mg/usr/mg", then press return.
 - c. Type "mg768 768/ah64.flt", then press return.
 - d. Pull down the "ANIMATION" menu bar and select the "Set Up" command; the name of the set up file is called "768_ani_flt.doc".
 - e. Close the "World View" window and then select the "Lead Helicopter View"; the name of the link file for the cockpit instruments is called "radar_alt.lnk".
 - f. Pull down the "ANIMATION" menu and select the "Full Screen" command, and then select "Ethernet" command to start the simulation.

6.6.2 Creating model

In general, a geometric model can be created by using MultiGen's modeling capabilities. Please consult *Software Systems MultiGen, Modeller's Guide* (September, 1988) on how to create and edit 3D models. A special feature on creating terrain surface from elevation data of DMA DTED is added for convenience which will be described in detail in section 6.6.4.

6.6.3 The Animation menu

Set Up brings up a temporary window and asks for the set-up file name. A set-up file contains the identification number (group ID) for all the animated objects. Information of the set-up file is echoed back in a separated window. Click "DONE" button if the user satisfies with all the information. Next, a window for the world view is displayed on the top right corner of the screen.

World View is used to re-opened the world view window if this window is closed during the simulation. This window presents the god's eye view of the whole simulation. Changing eye point

location, zooming and panning are allowed at any time and their operations follow the MultiGen conventions.

Moving Camera is used to specify a camera location of a window to display what that camera would see at that particular location. The user has an option to select a camera location from a list of 11 moving camera locations. The list of moving camera locations is popped up on the screen when this command is selected. Usually, the actual location for the moving camera view is at the graphics origin of the animated object, such as the helicopter, truck, etc. except the sensor's view and the lead helicopter observer's view. The camera location of sensor's view and lead helicopter observer's view are defined by user using the command "Def Sensor Loc." and "Def Observer Loc.", respectively. No special operations like zooming or panning are allowed but the window itself can be relocated to some other position on the screen by the user.

Fix Camera is used to specify a camera location inside the gaming area for monitoring the progress of the simulation. The user can specify up to 10 stationary camera locations inside the gaming area. The fix camera window will not be opened until the camera location is defined. The fix camera location is specified by defining the camera location inside the gaming area and its initial line of sight direction. A camera window pops up on the screen to display the view after defining those two points. The user can use the roll dial on the view control area to change the line of sight direction.

Head Up Display is a toggle switch to turn the Head-Up Display on or off. The Head-Up Display is displayed on the top of the pilot's view and within the same window. This display simulates what the pilot would see if he puts on the night vision goggles. It contains the symbology of helicopter's AGL altitude, compass direction, torque, airspeed, velocity vector and a 30 by 40 field of view area.

Ethernet is used to signal the beginning of the simulation. This command can only be selected after the completion of the "Set Up" command. "Ethernet" command puts the Views software in communication mode with a stand-alone file called "phase2.float" which contains all the information as specified in the file called "message.h" which is listed in Appendix D.

Stop Animation is used to stop the animation.

Resume Animation is used to resume the animation if it is stopped previously using the "Stop Animation" command.

Define Flight Path is used to plan a route for a helicopter mission. The route is picked by placing the cursor at the desired locations on a gaming area and clicking the mouse button. At each selected location, you are prompted to enter the altitude above the terrain surface, the airspeed, and the heading of the helicopter. After completing the selection of the route, the Views software saves all the information into a user specified file for playing back the mission using the command "Fly Thru Path". A terrain surface has to be displayed on the screen first before selecting this command. In order to get a better accuracy for picking stations along the route, choose the display option of orthographic view instead of perspective view. This is done by turning on the "ortho flag" located in the view control area in the same window. When the "Edit Control Window" appears at the upper left corner of the screen saying "Pick a point", the user can move the mouse to the desired location and click the left button. A solid line connecting all stations is displayed on the screen to indicate the route. The user can reject the last picked station by clicking the "UNDO" box in "Edit Control Window". Click the "DONE" box to terminate the "Define Flight Path" command.

Fly Thru Path is used to animate the mission specified by "Define Flight Path". This command allows the user to see the surrounding area along the selected route through the pilot's view window. The "Set Up" command needs to be selected before submitting this command. The data base file must contain the same terrain surface and a helicopter model. The user can specify the flying distance between frames. Specifying equal flying distance makes the flight (animation) a lot smoother. A scaling factor can be specified if the elevation information between the flight path data file and the terrain model are not in the same scale. Select "DONE" button if all values are set.

Freeze Frame is used to halt the animation temporarily and display the current frame on the screen. Once this command is selected, the Views software stops receiving data. The animation can be resumed again by pressing the left mouse button.

Single Tick is a toggle switch which can put the animation into either frame by frame mode or continuous mode. When the animation begins, the Views software displays the graphics continuously without interruption. By selecting this command, the Views software displays one frame of information at a time and the user has to press the left mouse button in order to get the next frame.

Restr. Pilot View is a toggle switch which can turn the restricted pilot's view on or off. The restricted pilot's view is a viewing area looking out through the Head-Up Display. Those areas which are outside the restricted pilot's view are blocked away from the pilot's eye. Normally, the area which pilot can see with the night vision goggles is about 30 degrees vertical field of view and 40 degrees horizontal field of view.

Def Pilot's FOV is used to redefine the viewing size of the restricted pilot's view. A temporary window pops up on the screen showing the size of the current restricted pilot viewing area. The user can type in the new values to redefine the field of view in both horizontally and vertically. Select the "DONE" button to complete this command.

Def Sensor Loc. is used to define the location of the sensor relative to the pilot's eye location. The sensor location is defined in terms of distance (feet) horizontally and vertically away from the pilot's eye. Select the "DONE" button to complete this command.

Full Screen is used to display the current window using the whole screen of the console. It acts like a toggle switch to display the window in either full screen or the original size.

4D to Iris is used to send animation data from coral (Iris 4D) to manta (Iris 3120). By selecting this command on both machines, the animation data will be sent from coral through the network to manta, then same scene but different viewing perspective of the simulation can be displayed on two screens simultaneously.

Show Timing is used to display the time required to render a complete simulation tick. The time is displayed on the top right corner of the screen.

Control Display is to display the flight control devices of the lead helicopter. Once this command is selected, three windows pop up on the screen to display the symbolic representation of cyclic control device, collective control device and pedal respectively. These three window can be gone away by selecting this command again.

Def Observer Loc is used to define the location of the observer which is relative to the lead helicopter. The location is defined by entering the vertical distance, horizontal distance from the

tail, and the horizontal distance from the side of the lead helicopter. elect the "DONE" button to complete this command.

Draw Flight Path is used to display the flight path which is defined by using the "Define Flight Path" command. Each station is represented by a blue square at that location and stations are connected together by a solid line.

Change Way point is used to redefine the location of the way point during the execution of the "Fly Thru Path". One or more way point locations can be changed at one time. The procedure is very similar to "Define Flight Path" command. Once the modification is done, the "Fly Thru Path" command resumes and follows the modified flight path.

HUD is a toggle switch to turn on or off the display of a head-up display which is located on the top of the cockpit instruments panel. This display contains the symbology of helicopter's AGL altitude, compass direction, torque, airspeed, and velocity vector.

6.6.4 The Terrain menu

The Terrain menu is used to convert the raw elevation data from Defense Mapping Agency (DMA) Digital Terrain Elevation Data (DTED) into a 3D graphics polygonal representation of terrain surfaces. A preprocessing standalone program called "readdma" must be run before selecting any commands from the Terrain menu. The program "readdma" is used to extract elevation data from a 9-track distribution tape (DTED tape) and save the data into a binary data disk file. When this program is executed, it prompts for number of cells on this tape and the cell number(s) to be selected. The label of DMA DTED tape should provide enough information regarding the cell arrangement for user to do the selection. The data are then save into a file whose name is specified by the user and this file is referred as Post file by MultiGen. The "readdma" program is resided in the directory called "/f1/mg3.0/Mg/Dma". For further information on how to use this program and the Terrain menu, please consult *Software Systems MultiGen, DMA Terrain Conversion Option User's Guide*, (July, 1988).

In case the Iris workstation system does not have a half-inch tape drive, then a tape image file can be created by using a remote system with the half-inch tape drive. First of all, each record on the tape is extracted out and save in a temporary disk file. After extracting all data record for the same cell, then these disk files are concatenated together to form a tape image file. This file is then copied into the Iris system. When running "readdma" program, the user has to type in "readdma file" which signifies that a tape image file is used. The contents of those two script files are shown in the Appendices B and C.

7.0 APPENDICES

7.1 Glossary, Abbreviations

A ³ I	Army-NASA Aircrew/Aircraft Integration
DMA	Defense Mapping Agency
DTED	Digital Terrain Elevation Data
DFAD	Digital Feature Analysis Data
CDE	Cockpit Display Editor
FOV	Field of view
HUD	Head Up Display

ofmt	original format for the current selected data base file
ifmt	internal format of MultiGen kernel

Appendix A The content of makefile for creating MultiGen executable

```

#make file options: -DOPT_DMA

IMINC = ../Im
KERINC = ../Ker
DMAINC = ../Dma
A3IINC = ../a3i

OBJS = fltdbl.o fltlink.o fltcolor.o fltio.o fltid.o fltpage.o

TRDOBJ = fltrade.o

OPT_TRD_OBJS = fltlink.o fltrade.o
OPT_DMA_OBJS = fltlink.o fltdbl.o

MGTARGET = mgflt

$(OBJS) $(TRDOBJ): $(IMINC)/imstrucs.h $(KERINC)/mgfmt.h fltfmt.h
$(TRDOBJ): $(TRDINC)/mgtrade.h

touchtrd:
    rm -rf $(OPT_TRD_OBJS)
touchdma:
    rm -rf $(OPT_DMA_OBJS)
clean:
    rm -rf *.o
    cd $(IMINC); rm -rf *.o
    cd $(DMAINC); rm -rf *.o
    cd $(KERINC); rm -rf *.o
    cd $(A3IINC); rm -rf *.o

.c.o:
    $(CC) -I$(IMINC) -I$(KERINC) -I$(DMAINC) -I$(A3IINC) $(CFLAGS)
$(DFFLAGS) -c $<

$(MGTARGET): $(IMINC)/im*.o $(KERINC)/mg*.o $(OBJS) $(A3IINC)/*.o
$(DMAINC)/ter*.o
    cc $(IMINC)/im*.o $(KERINC)/mg*.o $(A3IINC)/*.o $(DMAINC)/ter*.o \
$(OBJS) $(LDFLAGS) -Zg -Zf $(CFLAGS) -o $(MGTARGET)
    echo

all:
    cd $(IMINC); make im DFFLAGS=" -DOPT_DMA"
    cd $(DMAINC); make dma DFFLAGS=" -DOPT_DMA"
    cd $(KERINC); make ker DFFLAGS=" -DOPT_DMA"
    cd $(A3IINC); make a3i DFFLAGS=" -DOPT_DMA"
    make $(MGTARGET) CFFLAGS="-O -lbsd -ldbm" DFFLAGS=" -DOPT_DMA"
    mv mgflt /f1/mg3.0/Mg/usr/mg/mgflt

```

```
it:
    make $(MGTARGET) DFLAGS="-DOPT_DMA"

alldbx:
    cd $(IMINC); make im CFLAGS=-g DFLAGS="-DOPT_DMA"
    cd $(DMAINC); make dma CFLAGS=-g DFLAGS="-DOPT_DMA"
    cd $(KERINC); make ker CFLAGS=-g DFLAGS="-DOPT_DMA"
    cd $(A3IINC); make a3i CFLAGS=-g DFLAGS="-DOPT_DMA"
    make $(MGTARGET) CFLAGS="-g -lbsd -ldbm" DFLAGS="-DOPT_DMA"
    mv mgflt /f1/mg3.0/Mg/usr/mg/mgflt
```

Note: The above makefile is made for the version which is running in IRIS 2500T. Hardware differences between Silicon Graphics 3000, and 4D/GT workstations necessitate slightly different drawing mode. To allow for these and other differences within only one set of source listings, conditional compiles (ifdefs) are employed. A conditional compile identifiers is added to the makefile. The following statement shows the required change :

Replace DFLAGS="-DOPT_DMA" by DFLAGS="-DOPT_DMA -DOPT_GT"

Appendix B The content of a script file for extracting data records from DMA's DTED distribution tape

```
# is used to read DMA tapes - Folda Gap
# using /dev/rmt12 -- 1600 bpi, no rewind
# rewind tape
mt -f /dev/rmt12 rewind
# read VOL
dd if=/dev/rmt12 of=VOL bs=80 count=1
##### cell #1
# read 1st group, HDR and UHL
dd if=/dev/rmt12 of=HDR1 bs=80 count=1
dd if=/dev/rmt12 of=UHL1 bs=80 count=1
dd if=/dev/rmt12 of=/dev/null bs=1 count=1    # skip EOF mark
# read 2nd group, DSI and ACC
dd if=/dev/rmt12 of=DSI1 bs=648 count=1
dd if=/dev/rmt12 of=ACC1 bs=2700 count=1
# read data (1 files, 1201 records, 2414 bytes/record) ( 3 X 3 sec )
# read data (1 files, 601 records, 2414 bytes/record) ( 3 X 6 sec )
dd if=/dev/rmt12 of=cell1.data bs=2414 count=601
dd if=/dev/rmt12 of=/dev/null bs=1 count=1    # skip EOF mark
# read 3rd group, EOF1 and UTL1
dd if=/dev/rmt12 of=EOF1 bs=80 count=1
dd if=/dev/rmt12 of=UTL1 bs=80 count=1
# skip EOF mark
dd if=/dev/rmt12 of=/dev/null bs=1 count=1
```


Appendix C The content of a script file for concatenate the disk files into a single tape image disk file

```
set -x
cat VOL > dma.data1
cat HDR1 >> dma.data1
cat UHL1 >> dma.data1
cat eof.bin >> dma.data1
cat DSI1 >> dma.data1
cat ACC1 >> dma.data1
cat cell1.data >> dma.data1
cat eof.bin >> dma.data1
cat EOF1 >> dma.data1
cat UTL1 >> dma.data1
```


Appendix D message.h

```
typedef struct {
    long unused;
    int tick_no;

    float heli1_x, heli1_y, heli1_z;
    float heli1_yaw, heli1_pitch, heli1_roll;
    float heli1_speed;
    float heli1_altitude;
    float heli1_vvectorx, heli1_vvectory, heli1_vvectorz;
    float heli1_earth_vz;
    float heli1_torque;
    float heli1_explosion;

    float heli2_x, heli2_y, heli2_z;
    float heli2_yaw, heli2_pitch, heli2_roll;
    float heli2_speed;
    float heli2_altitude;
    float heli2_vvectorx, heli2_vvectory, heli2_vvectorz;
    float heli2_earth_vz;
    float heli2_torque;
    float heli2_explosion;

    float truck1_x, truck1_y, truck1_z;
    float truck1_yaw;
    float truck1_explosion;

    float truck2_x, truck2_y, truck2_z;
    float truck2_yaw;
    float truck2_explosion;

    float truck3_x, truck3_y, truck3_z;
    float truck3_yaw;
    float truck3_explosion;

    float truck4_x, truck4_y, truck4_z;
    float truck4_yaw;
    float truck4_explosion;

    float truck5_x, truck5_y, truck5_z;
    float truck5_yaw;
    float truck5_explosion;

    float truck6_x, truck6_y, truck6_z;
    float truck6_yaw;
    float truck6_explosion;

    float missile1_x, missile1_y, missile1_z;
    float missile1_yaw, missile1_pitch;
```

```
float missile1_explosion;

float missile2_x, missile2_y, missile2_z;
float missile2_yaw, missile2_pitch;
float missile2_explosion;

float missile3_x, missile3_y, missile3_z;
float missile3_yaw, missile3_pitch;
float missile3_explosion;

float missile4_x, missile4_y, missile4_z;
float missile4_yaw, missile4_pitch;
float missile4_explosion;

float missile5_x, missile5_y, missile5_z;
float missile5_yaw, missile5_pitch;
float missile5_explosion;

float missile6_x, missile6_y, missile6_z;
float missile6_yaw, missile6_pitch;
float missile6_explosion;

float missile7_x, missile7_y, missile7_z;
float missile7_yaw, missile7_pitch;
float missile7_explosion;

float missile8_x, missile8_y, missile8_z;
float missile8_yaw, missile8_pitch;
float missile8_explosion;

float missile9_x, missile9_y, missile9_z;
float missile9_yaw, missile9_pitch;
float missile9_explosion;

} animate_message;
```

Appendix E animate.menu

ANIMATION
Set Up
World View
Moving View
Fix View
-Head Up Display
Ethernet
Stop Animation
Resume animation
-Define Flight Path
Fly Thru Path
-Freeze Frame
Single Tick
-Restr. Pilot View
Define Pilot's FOV
Def Sensor Loc.
-Open Cockpit File
Full Screen
-4D to IRIS
-Show Timing
-Control Display
Def Observer Loc.
Draw Flight Path
Change Way Point
HUD

Appendix F animate.msg

7001:

No models exist ! Animation set up aborted !

/

7002:

No models exist ! Cannot create any views !

/

7003: Animation Set Up

Object Name	No.	Object Name	No.
<	> < >	<	> < >
<	> < >	<	> < >
<	> < >	<	> < >
<	> < >	<	> < >
<	> < >	<	> < >
<	> < >	<	> < >
<	> < >	<	> < >
<	> < >	<	> < >
<	> < >	<	> < >

</ DONE >

/

7004:

Invalid ID in use.

/

7005:

Needs to do set up first.

/

7006:

Enter set up filename : < >

/

7007: Restricted Pilot View's Field of View

Vertical Field of View : < >

Horizontal Field of View : < >

</ DONE >

/

7008: Location of The Sensor from the Pilot's Eye

Vertical Distance : < >

Horizontal Distance : < >

</ DONE >

/

7009:

Cannot find the standalone data file, communication abort !

/

7010: Location of The Observer from the Lead Helicopter

Vertical Distance : < >

Horizontal Distance from the tail: < >

Horizontal Distance from the side: < >

</ DONE >

/

7100: Moving Camera Location

Pick a point

/

8005: Flying through defined flight path

Enter flight path data file name : < >

Flying distance between each frame : < > ft.

Scale factor for the terrain's elevation : < >

</ DONE >

/

8006:

Open file failure.

/

8007:

Cannot draw the flight path. No flight path defined yet.

/

8008: Flight path definition:

Altitude above terrain : < > ft.

Air speed : < > knots

Helicopter heading : < > degrees

</ OK >

/

9001: Calculator

Enter your mathematical expression :

<

>

The result for the above expression : < >

Values for the variables :

U: < >

V: < >

W: < >

X: < >

Y: < >

$$Z: < \quad >$$

< OK >

/9002:

Enter FLIGHT database filename : < >
/

Annex C

Army-NASA Aircrew/Aircraft Integration Program

A³I

**Software Detailed Design Document:
Phase III Cockpit Design Editor**

prepared by

Teh-Ming Hsieh

December 1988

Table of Contents

1.0 INTRODUCTION.....	C-1
1.1 Identification.....	C-1
1.2 Scope.....	C-1
1.3 Purpose	C-1
2.0 RELATED DOCUMENTATION	C-3
2.1 Applicable Documents.....	C-3
2.2 Information Documents	C-3
3.0 REQUIREMENTS AND DESIGN APPROACH	C-3
3.1 Requirements and Rationale	C-3
3.2 Hardware Environment	C-4
3.3 Software Environment	C-4
4.0 DETAILED DESIGN DESCRIPTION	C-4
4.1 Organization.....	C-4
4.2 Unit Detailed Design.....	C-6
4.2.1 Interface Manager Subsystem.....	C-6
4.2.2 MultiGen Kernel Subsystem.....	C-6
4.2.2.1 Data Structure : mgfmt.h.....	C-7
4.2.2.2 Source Module : mgcom.c.....	C-8
4.2.2.3 Source Module : mgdesk.c	C-8
4.2.2.3.1 Constants and Variables	C-8
4.2.2.3.2 Inches/Feet Bubble Control : Enterfeet()	C-8
4.2.2.3.3 MG/A3I Bubble Control : Entermg()	C-9
4.2.2.4 Source Module : mgicnstr.c	C-9
4.2.2.5 Source Module : mgiconst.c.....	C-9
4.2.2.6 Source Module : mgicre.c.....	C-9
4.2.2.7 Source Module : mgidisp.c.....	C-9
4.2.2.7.1 Header File : cdefmt.h	C-9
4.2.2.7.2 Constants and Variables	C-10
4.2.2.7.3 Gouraud Shading Control : Drawfacesolid().....	C-10
4.2.2.7.4 Animation Control : Drawbead()	C-10
4.2.2.7.5 Viewing Control : IDR_setport()	C-10
4.2.2.7.6 Advanced Display Control : IDR_drawport().....	C-10
4.2.2.8 Source Module : mgiedit.c.....	C-10
4.2.2.8.1 Flight Path Construction : Faceundo.....	C-11
4.2.2.8.2 New Gauge Reference Line Control : CDE_gfface ...	C-11
4.2.2.8.3 Define Flight Path Control : ME_define_path.....	C-11
4.2.2.8.4 Define Camera Utility : ME_fix_camera.....	C-11
4.2.2.9 Source Module : mgifun.c	C-11
4.2.2.10 Source Module : mgipick.c.....	C-11
4.2.2.10.1 Pick Bead Utility : IP_pickscreen.....	C-11
4.2.2.10.2 Pick Bead Utility : IP_getsymdb.....	C-12
4.2.2.11 Source Module : mgmain.c.....	C-12
4.2.2.11.1 Header File : cdefmt.h.....	C-12
4.2.2.11.2 Constants and Variables.....	C-12
4.2.2.11.3 Kernel Control : main	C-13
4.2.2.11.4 MK Color Map : mgcolorinit	C-13
4.2.2.11.5 Delete Symdb Utility : ME_imanuever	C-13
4.2.2.11.6 Select from ID Utility : Selectmenu.....	C-13
4.2.2.11.7 Structure Menu Control : Strucmenu	C-13
4.2.2.12 Source Module : mgpage.c.....	C-13
4.2.3 Data Base Logic Subsystem.....	C-13

Table of Contents

4.2.4	Animation Kernel Subsystem.....	C-13
4.2.5	DMA Kernel Subsystem.....	C-13
4.2.6	CDE Kernel Subsystem	C-13
4.2.6.1	The CDE Data Structure : cdefmt.h.....	C-14
4.2.6.2	Interface Director : cdefile.c	C-14
4.2.6.2.1	Initializing the CDE : CDE_init.....	C-14
4.2.6.2.2	Database Scan and Execute : CDE_scan_hiera.....	C-16
4.2.6.2.3	Animation Control : CDE_transformation.....	C-17
4.2.6.2.4	Animation Initialization : CDE_animation_init.....	C-17
4.2.6.2.5	Animation Attribute modification : CDE_	
	modlnkdocw	C-17
4.2.6.2.6	Editable String Control : CDE_addgetstring	C-18
4.2.6.2.7	Multiple Bubble Control : CDE_mkbutton	C-18
4.2.6.3	Interface Director : cdefunc.c.....	C-18
4.2.6.3.1	Initializing the Binary Tree : UDF_buildtree.....	C-19
4.2.6.3.2	Evaluate the Binary Tree : UDF_eval	C-19
4.2.6.4	Instrument Editor : cdegauge.c.....	C-19
4.2.6.4.1	Mathematics Background.....	C-19
4.2.6.4.2	Top Level Menu Control : GAU_newgauge	C-19
4.2.6.4.3	Gauge Paste Control : GAU_paste_gauge	C-21
4.2.6.4.4	Writemask Control : GAU_gaugemask,	
	GAU_normalplanes.....	C-21
4.2.6.4.5	Z-axis Alignment : GAU_refangle	C-21
4.2.6.4.6	Translation Animation X-Axis Alignment :	
	GAU_transadjust.....	C-21
4.2.6.4.7	Translation Animation X-Axis Alignment :	
	GAU_fltransadjust	C-22
4.2.6.4.8	Translation Animation X-Axis Alignment :	
	GAU_icoordadj.....	C-23
4.2.6.5	Advanced Display Editor : cdedisp.c	C-23
4.2.6.5.1	Initialization : ADV_init.....	C-23
4.2.6.5.2	Toggle Control Procedure : ADV_advdisp.....	C-23
4.2.6.5.3	Interface Control : ADV_adspsetup	C-23
4.2.6.6	Parameter Linker : cdelink.c.....	C-24
4.2.6.6.1	Attribute Linking Control : LNK_linkit.....	C-24
4.2.6.6.2	Attribute Update Setup : LNK_modparam	C-26
4.2.6.6.3	Delete a Single Link : LNK_deletesymbdb	C-26
4.2.6.6.4	Remove Whole Link : LNK_rmlink	C-26
4.2.6.7	Color Handler : cdecolor.c.....	C-26
4.2.6.7.1	Initialization : CDCH_colorinit	C-26
4.2.6.7.2	Palette Control : CDCH_palette.....	C-26
4.2.6.7.3	Color Insertion : CDCH_insertcolor.....	C-27
4.2.6.7.4	Modify Color Control : CDCH_modcolor	C-27
4.2.6.7.5	Color Intensity Control : CDCH_hi_inten,	
	CDCH_lo_inten	C-27
4.2.6.8	Database Manager : cdepage.c.....	C-28
4.2.6.8.0	Database Field Elements Structure	C-28
4.2.6.8.1	Structure Pointer : CPG_getfieldtype	C-30
4.2.6.8.2	Field Format Control : CPG_getfield.....	C-30
4.2.6.8.3	String Field Control : CPG_getstring.....	C-30
4.2.6.8.4	Store Editable String to Database : CPG_putfield	C-31

Table of Contents

4.2.6.8.5	Verify the Editable String : CPG_movestr	C-31
4.2.6.8.6	GENERAL Field Handler : CPG_pagegeneral	C-31
4.2.6.8.7	Modify Attributes Window Control : CPG_mod_att..	C-31
4.2.6.8.8	New Attributes Window Control : CPG_newattr	C-32
4.2.6.8.9	Symdb Structure Init : CPG_newsymdb	C-32
4.2.6.8.10	Create New Link to Symdb Structure Tree : CPG_cresym.....	C-32
4.2.6.8.11	Attach Link to Parent Symdb Structure Tree : CPG_attach	C-32
4.2.6.8.12	Detach Link from Parent Symdb Structure Tree : CPG_detach.....	C-32
4.2.6.8.13	Set Parent Symdb : CPG_sel_set_attach.....	C-33
4.2.6.9	Database Manager : cdefmter.c.....	C-33
4.2.6.9.1	Selection Window : FMT_setup.....	C-33
4.2.6.9.2	Control Window Processes : FMT_setproc.....	C-33
4.2.6.10	Database Manager : cdeneig.c.....	C-33
5.0	Notes.....	C-33
5.1	Miscellaneous	C-33
5.2	Limitations.....	C-34
5.3	Future Directions.....	C-34
6.0	Users Guide.....	C-34
6.1	Introduction.....	C-34
6.2	Related Documentation.....	C-34
6.3	The Structure of CDE Hierarchical Database	C-34
6.4	CDE Menu Commands.....	C-35
6.4.1	Open Coord. File.....	C-35
6.4.2	New Gauge	C-35
6.4.3	Animation Init.....	C-36
6.4.4	Load Link File.....	C-36
6.4.5	Save Link File	C-37
6.4.6	Link Parameter.....	C-37
6.4.6.1	Chopper.....	C-37
6.4.6.2	Parameter	C-37
6.4.6.3	Function Handler	C-37
6.4.6.3.1	Linear Mapping Function.....	C-37
6.4.6.3.2	Periodic Mapping Function	C-37
6.4.6.3.3	Truncate Mapping Function.....	C-38
6.4.6.3.4	Linear Digit Mapping Function.....	C-38
6.4.6.3.5	Drum Digit Mapping Function.....	C-38
6.4.6.3.7	Natural Logarithmic Function.....	C-39
6.4.6.3.8	User Defined Function.....	C-39
6.4.6.4	Operation.....	C-39
6.4.6.4.1	Rotation.....	C-39
6.4.6.4.2	Translation	C-40
6.4.6.4.3	ADI.....	C-40
6.4.6.4.4	Pushbutton	C-41
6.4.6.4.5	Toggle Switch	C-41
6.4.6.4.6	Advanced Disp.....	C-42
6.4.6.4.7	Vertical Scale.....	C-42
6.4.6.4.8	Numerical LED.....	C-42
6.4.6.4	Writemask	C-42

Table of Contents

6.4.7 Color Palette	C-42
6.4.8 Insert Color	C-42
6.4.9 Modify Color	C-43
6.4.10 Mod Attribute.....	C-43
6.4.11 Data Format O/P	C-43
6.4.12 ADSP Setup	C-43
6.4.13 Advanced Disp	C-43
6.4.14 Unlink All.....	C-43
6.5 Error Messages and Diagnostics	C-43
7.0 APPENDICES	C-43
A. Glossary of Terms, Acronyms, or Abbreviations	C-43
B. Sample Displays	C-43

1.0 INTRODUCTION

1.1 Identification

This document establishes the requirements and detailed design of the Cockpit Design Editor Computer Software Configuration Item (CSCI), which forms a part of the A3I Computer Program System. Descriptions of the detailed processing requirements, structure, I/O, and control are provided for each lower level Computer Software Component (CSC), unit, or function contained within the CSCI.

1.2 Scope

This document describes the framework, function, and operation of the Cockpit Design Editor developed during Phase III of the A³I Program. It is assumed that the reader has some prior experience and understanding of how MultiGen works (i.e. mouse activated commands, pull-down windows, dialog boxes, windows).

1.3 Purpose

The software described herein provides designers with the necessary tools to develop next generation cockpits. The Cockpit Design Editor (CDE) operates within MultiGen, a commercially available CAD modelling package developed and owned by Software Systems, located in San Jose, California. The CDE is implemented as a subsystems of MultiGen and was developed by A³I at NASA Ames Research Center.

The effectiveness of a designing tool is determined by its ease of use and power of validation. The CDE builds upon the MultiGen user interface and its inherent intuitiveness. The cockpit designer is provided a

The basic operating and programming techniques of MultiGen are well documented in the Software Systems MultiGen manuals. This document deals with the CDE software only. Figure 1 shows the CDE's internal modular structure. Currently, six software modules have been developed for the CDE.

1. **Interface Director** - This is the central control module of the CDE kernel. It performs several functions: initializes CDE kernel during program boot-up, controls data and commands flow when a CDE menu has been selected, reacts to an animation request from the MultiGen display module, and converts an A3I datafile to MultiGen i-format file.

2. **Instrument Editor**- Provides a set of construction tools enabling the user to easily model conventional cockpit designs in 3-D. The Instrument Editor's built-in standard 3D display/control library supports switches; dial, vertical scale, and knob type gauges; and vector fonts. For instruments not included in the standard library, the operator can design the desired instrument using the standard MultiGen modelling utilities..

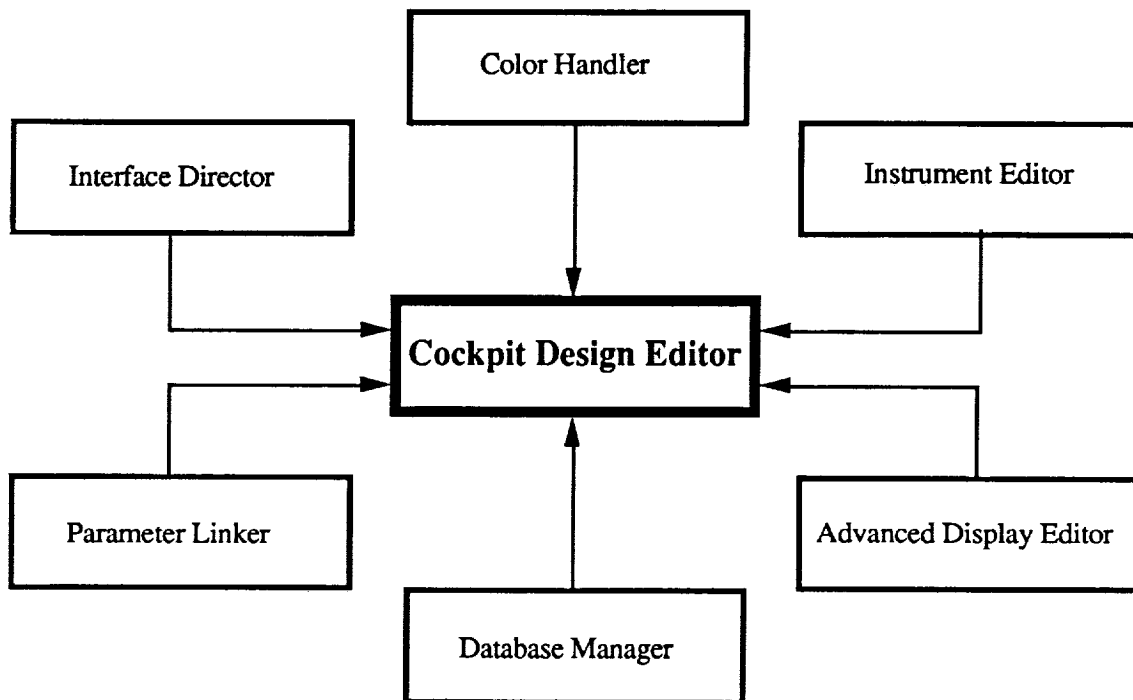


Figure 1. The modular structure of the Cockpit Design Editor.

3. **Advanced Display Handler** - Gives the user a gateway for constructing a glass cockpit type multifunction display, which would be difficult, if not impossible, to generate by using the standard Instrument Editor or MultiGen features alone. Two advanced displays, Perspective Display and Contour Display, are currently running under the Advanced Display Handler.
4. **Parameter Linker** - Provides an interactive environment for the user to evaluate the performance of the instruments that he or she just created or modified, in the "real" world. The Parameter Linker gives the user full control to define the gauge animation type, model parameter selection, and mathematical function handler. This visual interface allows the user to predict the actual reactions for all instruments during the flight (execution).
5. **Color Handler** - Manages the color related CDE operations, such as lighted switch and warning panels, etc.. The "writemask" utility provided by the Color Handler allows layering of the images on the same instrument, such as ADI, drum counter, and heading tape.
6. **Database Manager** - Provides an interface for the designer to evaluate cockpit prototypes in a non-MultiGen environment. A hierarchical descriptive database interface allows user to define special properties for each instrument. Designer can link this database to other computer models for on-line or off-line design analysis. Two graphical database formats are also provided for other applications.

2.0 RELATED DOCUMENTATION

2.1 Applicable Documents

Software Systems, *Software Systems MultiGen - Modeller's Guide, Interface Manager, MultiGen Kernel, Writing MultiGen DBL*, San Jose, CA, 1987.

Teh-Ming Hsieh, *Phase II Cockpit Display Editor Software - Software Component Description Document for the A3I Program*, 1987.

2.2 Information Documents

Yvon Gardan and Michel Lucas. *Interactive Graphics in CAD*, UNIPUB, New York, NY, 1984.

Franco P. Preparata and Michael Ian Shamos, *Computational Geometry - An Introduction*, Springer-Verlag, New York, NY, 1985.

Leendert Ammeraal, *Programming Principles in Computer Graphics*, John Wiley and Sons, West Sussex, England, 1986.

Nadia Magnenat-Thalmann and Danial Thalmann, *Computer Animation: Theory and Practice*, Springer-Verlag, Tokyo, Japan, 1985.

Silicon Graphics Inc., *IRIS User's Guide*, Volume I and II, Version 3.0, Mountain View, California, 1986.

Silicon Graphics Inc., *IRIS Programmer's Manual, Volume IB, System Calls and Subroutines*, Version 2.1, Mountain View, California, 1986.

3.0 REQUIREMENTS AND DESIGN APPROACH

3.1 Requirements and Rationale

The major development requirements for the CDE in the Phase III period are:

- To make the CDE compatible with the new Flight database.
- To design a descriptive database interface which allows user to specify physical meaning for each instrument.
- To provide a non-programmer interactive graphic interface for designers to prototype cockpits in 3D.
- To upgrade animation features for the helicopter dynamics and system models.
- To improve the capabilities to integrate, and to interact with the "Glass Cockpit" style advanced displays.
- To develop a graceful, intuitive mechanism for the user to link instruments and displays to the system models.
- Avoid duplication on program development and training.

3.2 Hardware Environment

The suggested minimal equipment configuration for running the CDE and for code development is a Silicon Graphics IRIS 2500T color system, with 8 MB of memory, 32 1024 x 1024 bit-planes with 16 bit Z-buffer, and 120 MByte storage capacity

Ideally, the best platforms for running and developing code for the CDE are SGI's IRIS 4D series computers. The new IRISes, based on a totally different architecture than its predecessors, are considerably quicker both in terms of CPU speed and geometric rendering. The same memory, bit-plane, Z-buffer and storage configuration as listed for the IRIS 2500T are also recommended for the IRIS 4D.

3.3 Software Environment

Elements of the IRIS software environment most important to the CDE Phase III effort are:

- The IRIS Graphics Library II (GL2), a general-purpose, easy-to-use graphics library.
- the Software Systems' Interface Manager, a mouse and window oriented user interface library.
- TCP/IP, an industry standard Ethernet network protocol,
- Network File System (NFS), a remote file access facility,
- The UNIX V operating system with Berkeley (4.3 bsd) extensions. C compiler, and a development/debugging environment integrated with the C compiler.
- System V and Berkeley utilities for source-code control (sccs and rcs), program debugging (adb, sdb and dbx), and code development (make, lint, lex, yacc, awk and others), in addition to more than three hundred utilities available on most recent versions of System V and 4.3 bsd UNIX.

4.0 DETAILED DESIGN DESCRIPTION

4.1 Organization

As shown in Figure 2, the CDE software is integrated within the MultiGen system and Views software. MultiGen is an interactive modeling system for creating, editing, and viewing 3D data bases for visual simulation. The purpose of the Views software is to generate a set of tools for creating the simulation world. The original MultiGen system is composed of three separate software

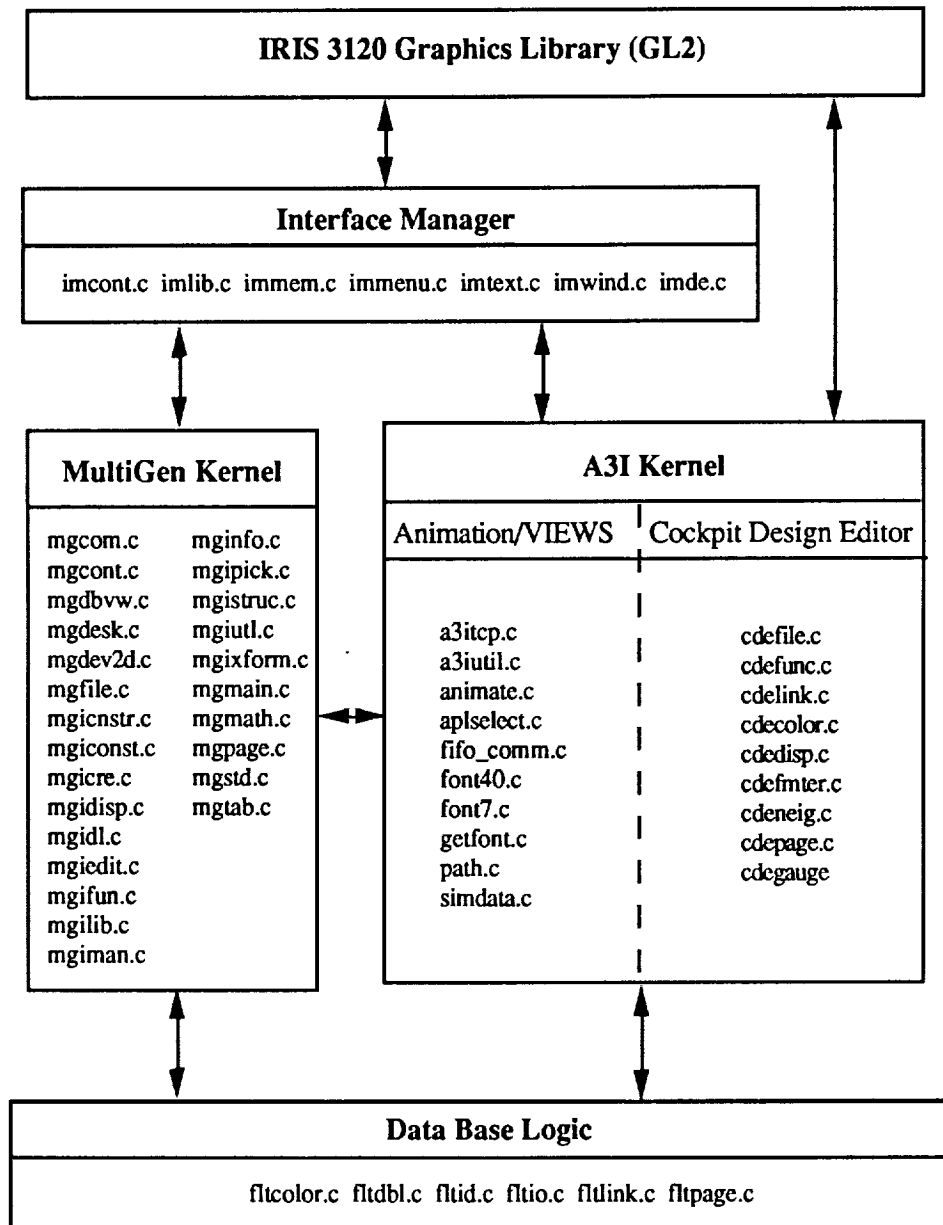


Figure 2. The Integration of CDE and Other Software Systems.

subsystems (i.e. the Interface Manager Subsystem (IM), the MultiGen Kernel Subsystem (MK) and the Data Base Logic Subsystem (DBL) that are linked together to make an executable MultiGen program. The CDE software is built on top of the original MultiGen system as a subsystem. Similarly, Views is also built on top of MultiGen but consists of two subsystems: Animation

Kernel Subsystem and DMA Kernel Subsystem. All software systems discussed herein are implemented on a Silicon Graphics IRIS 3120 workstation. The Graphics Library of the IRIS 3120 is a set of graphics and utility routines that provide high- and low- level support for graphics.

- The Interface Manager Subsystem is a collection of procedures that support a mouse and window oriented user interface. The subsystem accepts all the commands from the user and passes the information to the appropriate subsystem.
- The MultiGen Kernel Subsystem consists of a programming environment and a graphics editor. The programming environment provides procedures that can be called to create or manipulate MultiGen's internal format and parts of the user interface, and to perform other utility functions.
- The Data Base Logic Subsystem (DBL) provides data base independence in MultiGen. Its functions are to access an "ofmt" for the MultiGen Kernel, to handle user interface issues such as coordinate displays, terminology and modes, and to provide special purpose menu functions for an "ofmt". It is implemented as a series of low level procedures that are called from the MultiGen Kernel via the DBL linkage.
- The Animation Kernel Subsystem provides a facility to view the A³I Mission Simulation in different viewing perspectives using 3D graphics. Windows are displayed on the screen to present the views. The Animation Kernel Subsystem obtains simulation data from the Executive program through inter-process communication pipeline or reading from a data file.
- The DMA Kernel Subsystem is used to extract elevation data from any standard Level 1 DMA's DTED tape and convert it into shaded 3D representations of terrain surfaces. The generated terrain surface is saved into a data base file for later editing.
- The CDE Kernel Subsystem provides tools to construct, animate, and evaluate cockpit prototypes. The CDE Kernel Subsystem closely interacts with other Kernels through strictly defined procedure calls and global data structures linkage.

4.2 Unit Detailed Design

4.2.1 Interface Manager Subsystem

The Software Systems' Interface Manager Subsystem (IM) is a collection of C procedures that support a mouse and window oriented user interface on the Silicon Graphics IRIS workstation. It provides the interface between the user and the application program. The Interface Manager imposes an architecture on the application program that is appropriate for the random sequence input from the user. Detail information on describing the Interface Manager Subsystem's environment and procedures can be found in *Software Systems MultiGen, Programmer's Guide to the Interface Manager*, Release 1.1, September, 1987. For detailed IM modification please refer to Views SDDD.

4.2.2 MultiGen Kernel Subsystem

Software Systems' MultiGen Kernel Subsystem is composed of 25 source modules. As in the IM, externally callable procedures are preceded by a two to four capital prefix and an underscore. A brief description for each module listed below.

<u>Module</u>	<u>Prefix</u>	<u>Purpose</u>
mgcom.c	COM_	Communication of variables common to DBL and kernel.
mgcont.c	AC_	MultiGen auxiliary controls.
mgdbvw.c	DBVW_	Handles controls which associated with a MultiGen view window.
mgdesk.c	DH_	MultiGen desktop control.
mgdev2d.c	DEV2_	MultiGen 2-D device independence file.
mgfile.c	FM_	MultiGen file management routines.
mgicnstr.c	MCN_	MultiGen low level I format construction routines.
mgiconst.c	MC_	MultiGen I format construction routines.
mgicre.c	MCR_	Complex object creation routines.
mgidisp.c	IDR_	MultiGen I format pick and display.
mgidl.c	ID_	MultiGen display list utilities.
mgiedit.c	ME_	MultiGen I format edit routines.
mgifun.c	GF_	MultiGen graphics primitives manipulation functions.
mgilib.c	IL_	MultiGen I format bead management routines.
mgiman.c	MN_	MultiGen top level maneuver routines.
mginfo.c	MIF_	MultiGen timebomb information.
mgipick.c	IP_	MultiGen I format pick procedures.
mgistruc.c	ISD_	I format structure display.
mgitul.c	IU_	MultiGen I format utility routines.
mgmain.c	MA_	MultiGen kernel main program.
mgmath.c	ML_	MultiGen mathematics library functions.
mgpage.c	PA_	MultiGen ofmt page editor.
mgtab.c	TAB_	MultiGen auxiliary input device file.
mgstd.c	SS_	MultiGen standard header access routines.
mgixform.c	XFLL_	Instance transformation tools module.

Table 1-1. MultiGen Kernel Source Modules.

Detailed information regarding MK please refer to *Software System MultiGen, Programmer's Guide to the MultiGen Kernel*, Release 2.0, 1987. Some files which have been tailored for A3I's simulation will be discussed in this section. Source listing for MK can be found at the directory "/f1/mg3.0/Mg/Ker" of orca.

4.2.2.1 Data Structure : mgfmt.h

The global collection of MK variables and constants are defined in file "mgfmt.h". Five variables are added to "ibead" structure in "mgfmt.h" for A3I applications.

<u>Variable</u>	<u>Purpose</u>
int a3iflag	TRUE if current ibead is an Views animated object; FALSE if otherwise.
int cdeflag	= 0 current ibead is not a CDE animated object. MSB: TRUE-turn on CDE writemask; FALSE if not.

int gdbinfo	LS 7-bit: represent the CDE mapping function. pointer to dynamics/system model parameters structure.
int cdeinfo	pointer to CDE animation adjustment structure.
int syminfo	pointer to CDE descriptive database structure.

4.2.2.2 Source Module : mgcom.c

Two externally referenced variables "COM_internal_resol" and "COM_internal_resol_inv" are modified for higher database resolution. Listed below are the available resolutions:

```

/* x = (3/8) / 12.0 = 0.03125;    1/x = 32.0 */
/*MLTYPE COM_internal_resol = 32.0;
MLTYPE COM_internal_resol_inv = 0.03125;
*/

/* x = (1/64) / 12.0 = 0.001302083;  1/x = 768.0 */
MLTYPE COM_internal_resol = 768.0;
MLTYPE COM_internal_resol_inv = 0.001302083;

/* x = (1/128) / 12.0 = 0.000651041;  1/x = 1536.0 */
/*
MLTYPE COM_internal_resol = 1536.0;
MLTYPE COM_internal_resol_inv = 0.000651041;
*/

/* x = (1/256) / 12.0 = 0.000325520849;  1/x = 3072.0 */
/*MLTYPE COM_internal_resol = 3072.0;
MLTYPE COM_internal_resol_inv = 0.000325520849;
*/

```

User selects the desired resolution and comment out others. The sample listing above showing user picking the 768-internal-resolution equal to 1-foot.

4.2.2.3 Source Module : mgdesk.c

The mgdesk module is modified to handle A3I unit exchange routines. The modification includes two global variables and two local reference procedures.

4.2.2.3.1 Constants and Variables

<u>Variable</u>	<u>Purpose</u>
#define FEETBUTTONS	added feet/inch bubble in control window
#define MGBUTTON	added mg/cde bubble in control window
int DH_Displaycoordtype	= 0 current displayed coordinates are in feet. = 2 current displayed coordinates are in inches.
int DH_Displaycdetype	= 0 normal MultiGen database logic. = 1 switch to CDE database logic.

4.2.2.3.2 Inches/Feet Bubble Control : Enterfeet()

The Enterfeet function sets up the variable DH_Displaycoordtype and updates the tracking coordinates in the tracking window.

```
/* called when "inches/feet" bubble control hit */
int Enterfeet ( w, c, button, oldc, drawflag )
windowpt w;      /* pointer to control window */
controlpt c;      /* not used */
int button;       /* id for the selected bubble */
controlpt oldc;   /* not used */
int drawflag;     /* not used */
```

4.2.2.3.3 MG/A3I Bubble Control : Entermg()

The Entermg function sets up variable DH_Displaycdetype which controls the database command logic.

```
/* called when "MG/A3I" bubble control hit */
int Entermg ( w, c, button, oldc, drawflag )
windowpt w;      /* not used */
controlpt c;      /* not used */
int button;       /* id for the selected bubble */
controlpt oldc;   /* not used */
int drawflag;     /* not used */
```

4.2.2.4 Source Module : mgicnstr.c

All of the modifications in this module are related to add an extra argument in IL_addivertex function calls. Refer to module mgilib.c for detailed information.

4.2.2.5 Source Module : mgiconst.c

The MC_mouse2vtx function in this module has been modified for the "define flight path" utility. During this mode, multiple surfaces are involved for coordinate computations. When the user selects a point the tracking plane function PAT_trackplane is called to locate the correct terrain surface, and setup a new plane equation for coordinate computation.

4.2.2.6 Source Module : mgicre.c

All of the modifications in this module are related to the extra argument in IL_addivertex function calls. Refer to module mgilib.c for detailed information.

4.2.2.7 Source Module : mgidisp.c

Major modifications are made in this module for the A3I animation facilities. Please study the code carefully before making changes.

4.2.2.7.1 Header File : cdefmt.h

Header file cdefmt.h is included in this module for global reference to CDE defined constants.

4.2.2.7.2 Constants and Variables

<u>Variable</u>	<u>Purpose</u>
#define SGIDLDRAW	changed to FALSE to enable immediate draw mode.

4.2.2.7.3 Gouraud Shading Control : Drawfacesolid()

This function has been modified for the A3I Gouraud shading utility. The modified program is activated by shading flag IDR_shade. When IDR_shade is TRUE, the program will first look for vertex color "vcolor", if it exists. Gouraud shaded surfaces are constructed according to the vertex colors, otherwise, a flat shaded surface defined by face color is displayed.

4.2.2.7.4 Animation Control : Drawbead()

To animate objects in the MultiGen environment it is necessary to track all the beads all the time. The best place to insert these animation control commands is in the Drawbead function, since it will recursively scan all beads in order. The modifications are inserted at the beginning and ending sections of the drawing loop :

- 1). MultiGen original matrix based animation operation are disabled.
- 2). Check for bead->a3iflag, if TRUE then program will call the ANA_transformation to setup world object animation.
- 3). Check for bead->cdeflag, if TRUE then program will call the CDE_transformation to setup CDE instrument animation.
- 4). Drawing loop.
- 5). Check for bead->cdeflag and bead->a3iflag, if TRUE then restore the corresponding graphics transformation matrix.

4.2.2.7.5 Viewing Control : IDR_setport()

This procedure is modified to create moving camera views and pilot views. The Control flag is the windowtype which is a member of IM's window structure. If the value of windowtype is greater than zero, than this window structure is pointed to a moving camera or pilot view window. For these windows, ANA_pilotview will overwrite the original procedure for computing and setting up the window viewing matrix. Refer to animate.c module for detailed information regarding ANA_pilotview function.

4.2.2.7.6 Advanced Display Control : IDR_drawport()

Code is added to this function for advanced display and head-up display. Control flags are ADVDSP_TYP for CDE advanced display and ANA_hud_flag for head-up display. The computations for lower left and upper right screen coordinates for the display region are done by the ANA_transformation and CDE_transformation functions. The handling function for CDE advanced display is ADP_advdisp of cdedisp.c, and for the head-up display it's ANA_hud_display of animate.c.

4.2.2.8 Source Module : mgiedit.c

Some of the modifications in this module are related to the extra argument in IL_addivertex function calls. Refer to module mgilib.c for detailed information. Other modifications are made for A3I applications.

4.2.2.8.1 Flight Path Construction : Faceundo

This procedure is modified to support the UNDO function of the "Define Flight Path" utility. During this mode, when the user issues an UNDO command, the waypoint stack pointer No_of_waypoint will decrement by one. The previous waypoint will be removed from the waypoint stack.

4.2.2.8.2 New Gauge Reference Line Control : CDE_gfface

This function is called to draw a dotted reference line when the user creates a new gauge.

```
int CDE_gfface ()
```

The CDE_gfface function takes no argument and three local functions GFvtx, GFundo, GFdone are called by CDE_gfface for control menu function handler.

4.2.2.8.3 Define Flight Path Control : ME_define_path

The ME_define_path is called to define a flight path on the 3D terrain. It will compute the coordinate and elevation for each selected point and prompt the user for the vehicle altitude, airspeed, and heading data at that waypoint.

```
ME_define_path ( change_way_point )
int change_way_point;    /* TRUE if modifying flight path, FALSE if creating
                           a flight path */
```

4.2.2.8.4 Define Camera Utility : ME_fix_camera

This function will draw a dotted line-of-sight reference line when the user defines a fixed camera view. Like the CDE_gfface function, ME_fix_camera takes no arguments, and the three local functions Fixvtx, Fixundo, Fixdone are called for by the control menu function handlers.

4.2.2.9 Source Module : mgifun.c

The function GF_coord2screen has been rewritten for compatibility with the new IRIS 4DGT graphics engine. Instead of using feedback mode, which is no longer supported by SGI, the A3I staff is using the getcpo command for maximum compatibility and processing speed.

```
int GF_coord2screen ( ip, iq )
icoordpt ip;    /* Input x,y,z world coordinate */
icoordpt iq;    /* Output x,y screen coordinate */
```

If input point is clipped out of the screen the FALSE will return and iq.x will equal minus one.

4.2.2.10 Source Module : mgipick.c

This module has been modified to handle additional CDE descriptive databases. When the user selects a screen object, depending on the DH_Displaycdetype, the MK graphics ID or CDE descriptive title will be displayed and flag them as selected.

4.2.2.10.1 Pick Bead Utility : IP_pickscreen

When condition control flag DH_Displaycdetype is TRUE, CDE symdb search routine IP_getsymdb is called to determine whether the mouse selected object is defined by the CDE database or not. If it is a defined object, its title will be displayed on the window and the object will be high-lighted, otherwise an error signal beep will be heard.

4.2.2.10.2 Pick Bead Utility : IP_getsymdb

This function locates the CDE database bead (symdb) based on current cursor position for top most window.

```
dialparampt IP_getsymdb ( topib, w, picklevel )
ibeadpt topib; /* top ibead pointer for current window */
windowpt w; /* current window structure pointer */
int picklevel; /* beadlevel to pick */
```

The IP_getsymdb procedure will first call Picksymdb with the ibead pointer and bead pick level. If Picksymdb returns FALSE, indicating there is no CDE database linked to the selected ibead, an error beep will sound. Otherwise, IP_getsymdb procedure will call Leveladjust with symdb pointer and picklevel. If Leveladjust returns FALSE, then the selected object is undefined in the pick level and an error beep will sound.

4.2.2.11 Source Module : mgmain.c

The global collection of MK variables and constants are defined in file "mgfmt.h". Five variables are added to "ibead" structure in "mgfmt.h" for A3I applications.

4.2.2.11.1 Header File : cdefmt.h

Header file cdefmt.h are included in this module for global reference of CDE defined constants.

4.2.2.11.2 Constants and Variables

<u>Variable</u>	<u>Purpose</u>
copterstrucpt Obj_infoptr	/* for VIEWS animation option */
int ANA_windowflag	/* flag for open animation window */
int ANA_standalone	/* animation standalone mode flag */
int APL_select_flag	/* APL flag for select area option */
int PAT_select_flag	/* APL 8-14-87 flag for flight path select option */
int VER_scale_flag	/* flag for vertical scale animation */
dialparampt Dial_infoptr	/* CDE descriptive database structure pointer */
dialanapt Dial_anapt	/* CDE animation run-time register structure */
int No_dial	/* number of defined CDE database */
int ADVDSP_TYP	/* default set to No advanced display */
int ID_Gouraud_shade	/* Gouraud shading flag */
int IDR_Zbuffer_flag	/* Zbuffer flag */
int AC_Elev_diff	/* fix camera elevation adjustment control */
int AC_old_y	/* v->lookfrom.y before change */
int IP_pickmode	/* Pick mode flag */
icoord IDR_adpll	/* lower-left corner coordinate of ADP */
icoord IDR_adpur	/* upper-right corner coordinate of ADP */
icoord ANA_hudll	/* lower-left corner coordinate of HUD */

```

icoord ANA_hudur      /* lower-left corner coordinate of HUD */
int ANA_hud_flag      /* flag for turn on the head-up-display */

```

4.2.2.11.3 Kernel Control : main

This function initializes MultiGen and turns over control to the Interface Manager. To make room for CDE color map the first argument of DBL_colormap has been changed to 256 in order to transfer 256 color indices from DBL to CDE. The DMA_init, ANA_init, and CDE_init functions are also added for A3I options.

4.2.2.11.4 MK Color Map : mgcolorinit

The color index 16 is reserved for the CDE writemask function. To avoid multiple color definitions at that index, conditional branch is added to prohibit MK from using that color index.

4.2.2.11.5 Delete Symdb Utility : ME_imanuever

If the DH_Displaycdetype is TRUE when the user selects the delete icon, LNK_deletesymdb will be called to unlink current selected symdb and below from link list.

4.2.2.11.6 Select from ID Utility : Selectmenu

If the DH_Displaycdetype is TRUE, when the user selects the "Select from ID" menu, FMT_str2sym will be called to find the corresponding symdb and flagg the object on the window.

4.2.2.11.7 Structure Menu Control : Strucmenu

If the DH_Displaycdetype is TRUE, when the user selects the Structure menu, the "set parent", "attach", and "detach" utilities are directed to the symdb structure handlers : CPG_sel_set_attach, CPG_attach, and CPG_detach.

4.2.2.12 Source Module : mgpage.c

To support CDE symdb database documentation utilities, the PA_docedpage function has been modified to carry database pointer (dblpt) into w->misc slot for the GENERAL document field.

4.2.3 Data Base Logic Subsystem

Flight database procedures are called from the Kernel through a DBL linkage module. The DBL linking module tightly links the Kernel and the Flight database together. For detailed information about the DBL subsystem please refer to Views SDDD.

4.2.4 Animation Kernel Subsystem

4.2.5 DMA Kernel Subsystem

4.2.6 CDE Kernel Subsystem

Refer to Figure 1. The CDE can be divided into five different functional modules, Interface Director, Instrument Editor, Advanced Display Editor, Parameter Linker, Color Handler and Database Manager. Each module contains one or more files, externally callable procedures in these files are preceded by a three or four letter file prefix and underscore. This allows the programmer

to easily locate the source file for a particular procedure from its name. Table 4.x shows the modules, source files, and prefix in CDE Kernel.

<u>Module</u>	<u>Source File</u>	<u>Prefix</u>	<u>Purpose</u>
Interface Director	cdefile.c cdefunc.c	CDE_ UDF_	CDE interface control manager user defined function handler
Instrument Editor	cdegauge.c	GAU_	Instrument editor manager
Advanced Display Editor	cdedisp.c	ADV_	Advanced Display edit and control
Parameter Linker	cdelink.c	LNK_	CDE animation manager
Color Handler	cdecolor.c	CDCH_	CDE color editor manager
Database Manager	cdepage.c cdefmter.c cdeneig.c	CPG_ FMT_ NGA_	CDE database interface manager Database conversion management Neighborhood analysis handler

TABLE 4.x CDE Kernel Subsystem Source Modules.

4.2.6.1 The CDE Data Structure : cdefmt.h

The definition of CDE data structures and defined constants are found in the file cdefmt.h. This file should be included in each application source program that uses CDE. The cdefmt.h will include the IRIS file stdio.h for the applications.

4.2.6.2 Interface Director : cdefile.c

The source file cdefile.c (Figure 3) is the CDE's overall interface manager, it responds to the CDE menu commands and gauge animation controls.

4.2.6.2.1 Initializing the CDE : CDE_init

The CDE_init is called when MultiGen put up the CDE menu:

```
CDE_init ()  /* Initialize CDE operations */
```

CDE_init setup the CDE environment by assigning the CDE message and menu file pointers, loading CDE color table, initializing CDE data structures and advanced display terrain data. Two files are opened by CDE_init:

cde.menu	Text menu resource file for the CDE pull-down menus that can be selected with the mouse to issue commands.
cde.msg	Text message resource file that defines the messages and templates which were used by the CDE.

Please refer to *Programmer's Guide to the Interface Manager* for detailed information on how to set-up menu and message files.

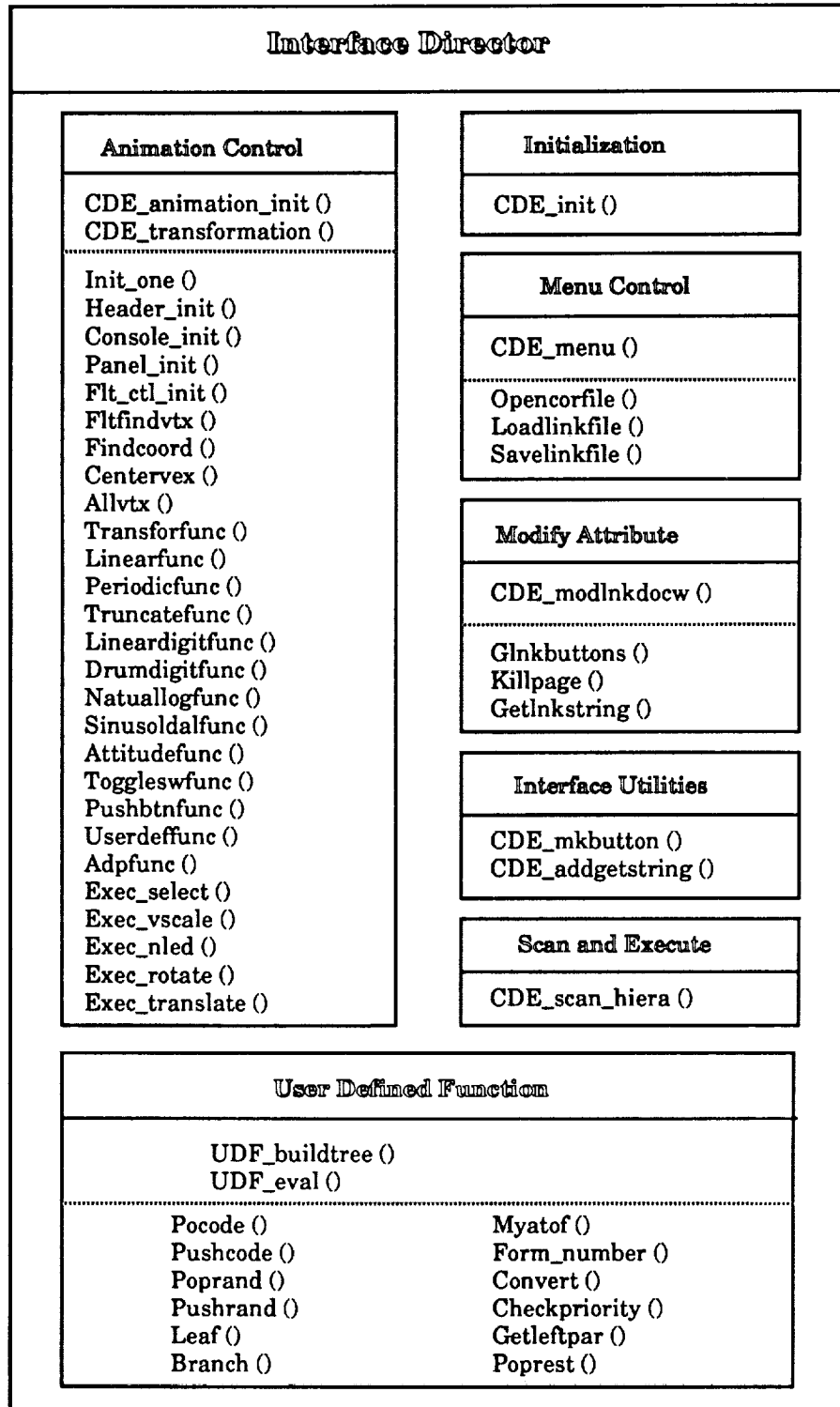


Figure 3. Procedures comprising each Interface Director function.

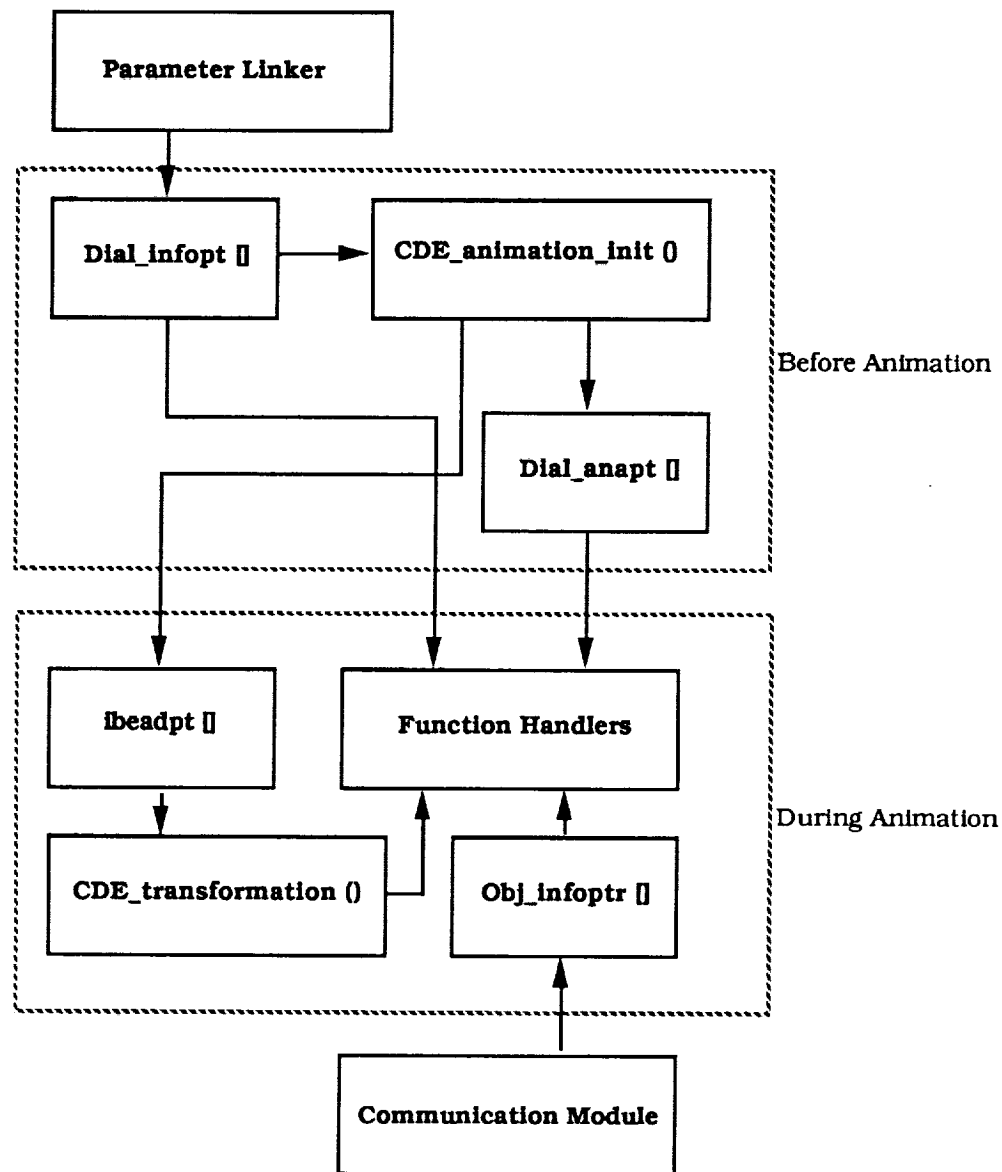


Figure 4. The Animation related procedures internal Structure.

4.2.6.2.2 Database Scan and Execute : CDE_scan_hiera

The CDE_scan_hiera procedure is called to scan all of the symdb bead tree below a given symdb and perform a given operation for each symdb.

CDE_scan_hiera (iarg, symdb, nextflag, func)

```

int iarg;           /* argument for operational function */
dialparampt symdb; /* scan this bead and tree below */
int nextflag;      /* control flag for recursive search */
int ( *func ) ();  /* operational function for each bead */

```

This is a general purpose database utility procedure which can be used by all CDE database applications. Usually the initial caller should set nextflag equal FALSE.

4.2.6.2.3 Animation Control : CDE_transformation

By decoding cdeflag for a given ibead, CDE_transformation will determine a graphics transformation matrix for the given ibead and tree below during the animation.

```

CDE_transformation ( ib )
ibeadpt ib;        /* animate this ibead and tree below */

```

Animation works by first finding the mapping function which computes the operation parameter from the given system parameter, the setup of the graphics transformation matrix and/or color assignment depends on its symdb operation flag. Figure 4 demonstrates the calling sequences for the animation related procedures.

4.2.6.2.4 Animation Initialization : CDE_animation_init

The CDE_animation_init is called when the programmer wants to initialize the CDE descriptive database and run-time animation registers.

```

CDE_animation_init ()      /* Animation Init menu item procedure */

```

CDE_animation_init calls the CDE_standalone to setup Views object structure, clear the old structure tree, and calls Init_one to establish database structure. The final step is to create a control level index array Gindex which will be used in the CDE Modify Attributes utility.

4.2.6.2.5 Animation Attribute modification : CDE_modlnkdocw

This routine is called when the user executes the CDE Modify Attribute command. It will display a list of instruments that are linked to the helicopter dynamics or system models. The user can select any item from the link list and interactively examine or edit its linking attributes. The changes will be reflected in the simulation immediately.

```

windowpt CDE_modlnkdocw ( de, msgno, maxlines, controlproc )
docedpt de;           /* message file pointer */
int msgno;            /* message ID */
int maxlines;         /* maximum lines in the text window */
int ( *controlproc ) (); /* function handler when SELECT button hit */

```

The CDE_modlnkdocw is not easy to understand nor to use. First, the user needs to setup a message file corresponding to the list structure:

```

struct {
    point delta; /* returned relative to lower left */
    point size;  /* returned as parameter size in X, Y */
}

```

```

        int    field; /* data field id defined by Getlnkstring () */
        int    ftype; /* button control type */
    } flist[ 30 ];

```

As shown below, displayed items are defined by the list structure. The flist.field will convert to the description string by procedure Getlnkstring; The Space between the left and the right brackets define the size of the viewing window; The flist.ftype is the button control flag.

4500: Sample Link List

```

ID          Purpose
[<flist[ 0 ].field>  <flist[ 1 ].field      >

```

]

```

</flist[ 2 ].ftype Select>

```

The CDE_modlnkdocw will first setup the list structure from message file, next, Getlnkstring is called for each animated instrument to fill the text array, and initialize button control procedure, than turn the control over to the window manager.

4.2.6.2.6 Editable String Control : CDE_addgetstring

This procedure is almost identical to the IDE_addgetstring except CDE_addgetstring sets up editable text with a color of WHITE instead of BLACK.

```

CDE_addgetstring ( w, string, maxlen, func )
windowpt w;          /* the document window pointer */
char *string;         /* the text string */
int maxlen;          /* maximum length of text */
int ( *func ) ();     /* called when user press RETURN */

```

The text string should be initialized to a NULL string or to a default string that will appear as selected text in the window when it is activated.

4.2.6.2.7 Multiple Bubble Control : CDE_mkbutton

This procedure allows the user to define a group of bubble controls which share the same bubble control function.

```

CDE_mkbutton ( bubble, w, func, ctrtype )
int bubble;          /* number of bubble controls want to create */
windowpt w;          /* window to add the control */
int ( *func ) ();    /* called as bubble state change */
int ctrtype;         /* control type should be BUBBLE */

```

CDE_mkbutton first calls the ICM_addfirstbubble to setup the first bubble control of the group, additional controls are added to the group by calling the ICM_addbubble procedure.

4.2.6.3 Interface Director : cdefunc.c

The User Defined Function should be part of the Animation Control, however, its unique characteristic makes it separate from other function handlers and require it to have a special file.

4.2.6.3.1 Initializing the Binary Tree : UDF_buildtree

4.2.6.3.2 Evaluate the Binary Tree : UDF_eval

4.2.6.4 Instrument Editor : cdegauge.c

The file cdegauge.c provides a non-programmer environment that allows the user to create standard gauges in 3-D space.

4.2.6.4.1 Mathematics Background

Before reading the source code, the programmer needs to understand the algorithms and program structure behind the operations. Basically, gauge pasting is a three steps problem:

- Interface Control Procedures - GAU_newgauge (), that set-up the menu driven user interface and controls the procedure calls in Instrument Editor.
- Axis Alignment Procedures - GAU_paste_gauge (), GAU_refangle (), and GAU_transadjust () procedures convert reference coordinates to world coordinate for each new instrument. The reference coordinate assumes all new gauges are drawn on the X-Y plane with the positive Z-axis as the direction of its normal vector. For every instrument to be created, an Attached Face and an Alignment Axis will be assigned by the user to compute the 3D transformation matrix (Figure 5 ,). The computation steps are as follows:
 1. Find the normal vector for the Attached Face
 2. Translate the Origin to the Center of the Instrument
 3. Align the Z-axis to the normal vector by calculating the X- and Y-axis rotation angles
 4. Rotate the Z-axis to let the Y-axis match up with the alignment axis
 5. Set-up the transformation matrix according to the center of the instrument and three rotation angles.
- Paste Instrument - This is done by individual "xxxshow" procedures, each procedure load the transformation matrix created in the earlier step and obtain the world coordinate by calling GF_transform_coord procedure with reference coordinate as argument.

4.2.6.4.2 Top Level Menu Control : GAU_newgauge

This procedure is called when user selects the CDE New Gauge menu item. It displays a list of the instruments that are supported by the build-in library, and prompt user to select one to paste on the instrument panel.

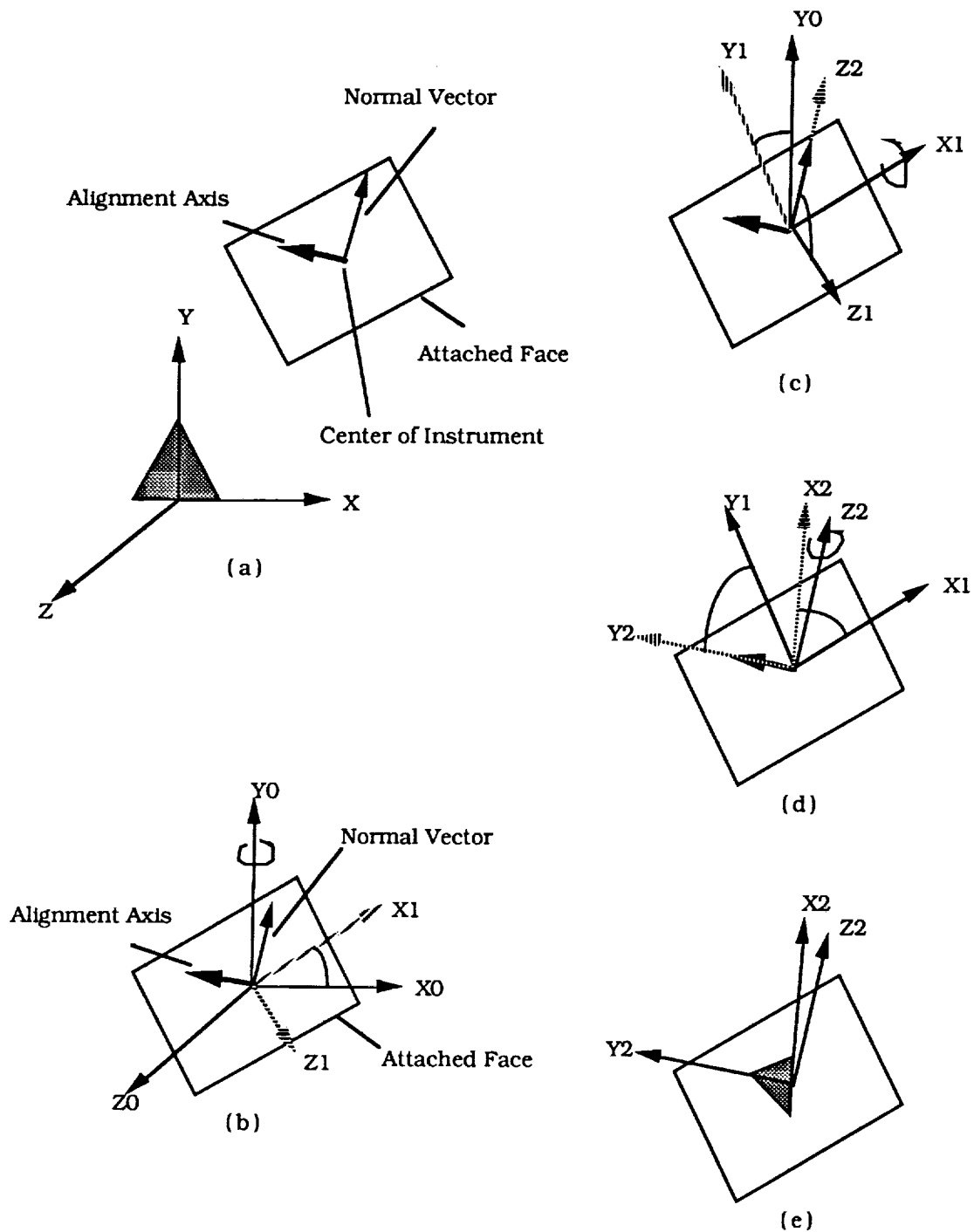


Figure 5. Axis Alignment procedures, (a) original 3D space, (b) (c) Align the Z-axis to the face normal vector, (d) Match the Y-axis to the alignment axis, (e) Transform the object to the new coordinate system.

4.2.6.4.3 Gauge Paste Control : GAU_paste_gauge

This procedure sets the paste gauge transformation matrix from pasted gauge center and alignment point and static variable Refstring which contains the attached face ID.

```
GAU_paste_gauge ( ctrp, refp )
icoord ctrp;    /* gauge center coordinate */
icoord refp;    /* alignment coordinate */
```

GAU_paste_gauge first calls DBL_id2i for the attached face ibead pointer, then calls GF_face_normal for the normal vector of attached face, then calibrates Z-axis with normal vector by calling GAU_refangle, and calls Gaugeadjust to match Y-axis with the alignment axis.

4.2.6.4.4 Writemask Control : GAU_gaugemask, GAU_normalplanes

These procedures are called when writemask operation is applied to a selected gauge. GAU_gaugemask will protect non-erasable from overlay by ordinary drawing routines and GAU_normalplanes is to reset writemask to original MultiGen writemask. Both procedure take no argument.

```
GAU_gaugemask ()
GAU_normalplanes ()
```

4.2.6.4.5 Z-axis Alignment : GAU_refangle

The GAU_refangle returns X- and Y-axis rotation angles that will align the given normal vector with Z-axis.

```
GAU_refangle ( i, j, k, thetax, thetay )
float i;      /* normal vector X-axis component */
float j;      /* normal vector Y-axis component */
float k;      /* normal vector Z-axis component */
int *thetax;  /* X-axis rotation angle ( x10 degree ) */
int *thetay;  /* Y-axis rotation angle ( x10 degree ) */
```

4.2.6.4.6 Translation Animation X-Axis Alignment : GAU_transadjust

The GAU_transadjust is called for translating related animation where all three axes needed to be align with attached face.

```
GAU_transadjust ( bead, fn, ip, vtx, thetay )
ibeadpt bead; /* bead pointer for attached face */
vector fn;    /* normal vector for the attached face */
icoord ip;    /* center or reference coordinate */
int vtx;      /* vertex order for the horizontal ( X-axis ) axis */
int thetay;   /* Y-axis rotation angle in order to match Z-axis to face normal */
```

This procedure also needs some detailed explanations of how it works. Normally, rotation related animations need only Z-axis to be aligned for the attached face. This routine aligns the X-axis with a user defined horizontal edge; doing this causes the Y-axis to align with the vertical axis automatically, that is, all three axes are aligned. First, GAU_transadjust forms a standard

horizontal vector ($v1$) from given edge vertex order (vtx). As shown in Figure 6, to form the second vector ($v2$) user have to compute $ipt3$ from ip and $thetay$, this is done by procedure `Rotate_y`. It is straightforward to obtain the X-axis adjustment angle ($trad$) from $v1$ and $v2$. The `GAU_transadjust` returns the angle in the unit of tenth of one degree.

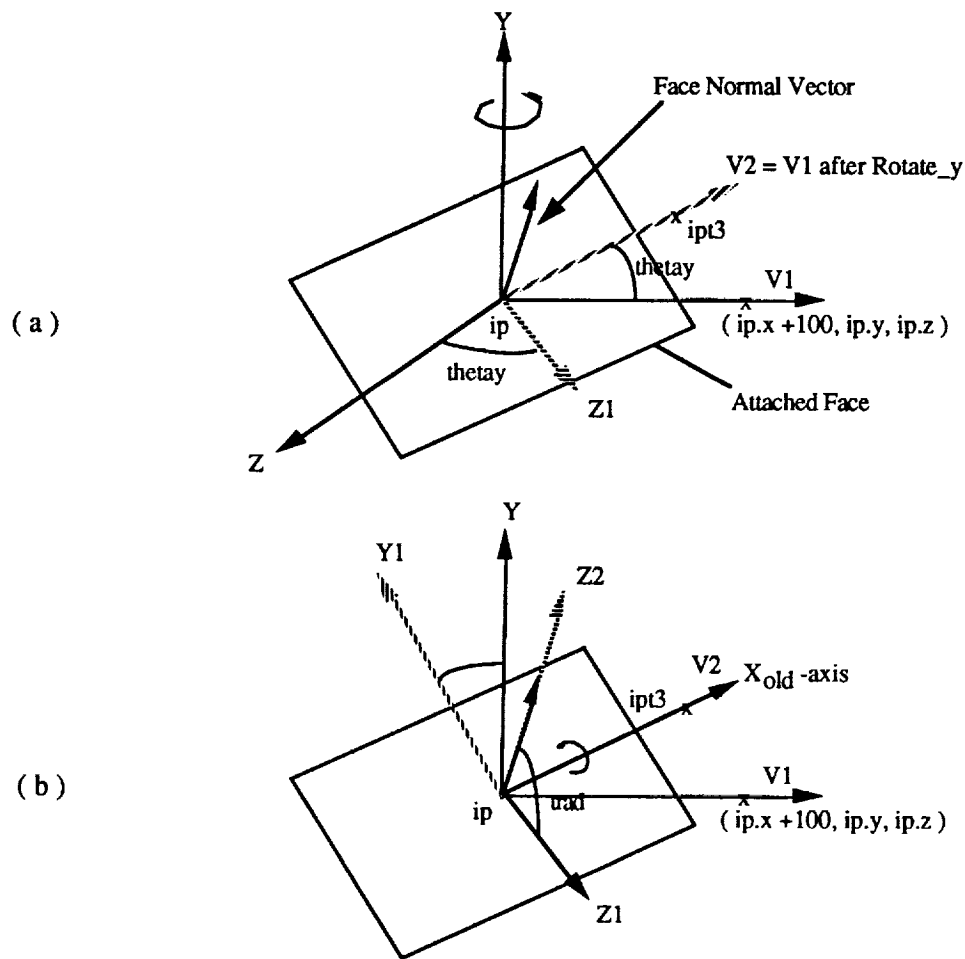


Figure 6. Translation Axis Alignment procedures (a) computes the new $ipt3$. (b) computes X-axis rotation angle $trad$ from $V1$ and $V2$.

4.2.6.4.7 Translation Animation X-Axis Alignment : `GAU_fittransadjust`

The `GAU_fittransadjust` serves the same purpose as the `GAU_transadjust` but uses different arguments to construct horizontal axis.

```
GAU_transadjust ( vtxpt, fn, ip, thetay )
  ivertxpt vtxpt;      /* leading vertex for the horizontal axis */
```

```

vector fn;          /* normal vector for the attached face */
icoord ip;          /* center or reference coordinate */
int thetay;         /* Y-axis rotation angle in order to match Z-axis to face normal */

```

Instead of converting the horizontal axis from ibead pointer and vertex order, GAU_transadjust directly form the axis from leading vertex pointer "vtxpt".

4.2.6.4.8 Translation Animation X-Axis Alignment : GAU_icoordadj

Like GAU_fltransadjust, the GAU_icoordadj serves the same purpose as the GAU_transadjust but use different arguments to form horizontal axis.

```

GAU_icoordadj ( ipt1, ipt2, fn, ip, thetay )
icoord ipt1;     /* starting vertex for the horizontal axis */
icoord ipt2;     /* ending vertex for the horizontal axis */
vector fn;       /* normal vector for the attached face */
icoord ip;       /* center or reference coordinate */
int thetay;      /* Y-axis rotation angle in order to match Z-axis to face normal */

```

GAU_icoordadj directly form the axis from two vertices ipt1 and ipt2.

4.2.6.5 Advanced Display Editor : cdedisp.c

This module contains one file, "cdedisp.c" which can be classified into Initialization, Toggle Control, and Interface Control (Figure 7). Currently, all displays (Vertical-type Perspective Display or VPD, 2-D Contour Display, and Radar Sensor Display) are hard coded and only VPD and Contour Display are fully functional. Nevertheless, the purpose of this module was to demonstrate how to interface CDE to the next generation of flight instruments.

4.2.6.5.1 Initialization : ADV_init

ADV_init setup the transformation matrix G_matrix and initialize the viewing window boundary points. It also format the DMA terrain elevation and 2-D contour arrays by calling Terrain_init and Contour_init procedures. The datafile that is opened for the initialization is: contour.dat for the binary terrain elevation data and contour.file for 2-D contour information.

4.2.6.5.2 Toggle Control Procedure : ADV_advdisp

ADP_advdisp select the viewing procedure defined by ADVDSP_TYP that enables different displays to be shown on the same viewing window sequentially.

```

ADP_advdisp ( iq1, iq2 )
icoord iq1;     /* viewing window lower left screen coordinate */
icoord iq2;     /* viewing window upper right screen coordinate */

```

4.2.6.5.3 Interface Control : ADV_adspsetup

ADV_adspsetup allows the user to change the viewing attitude angles and position during the animation. Currently, only the Vertical-type Perspective Display is functional for this utility.

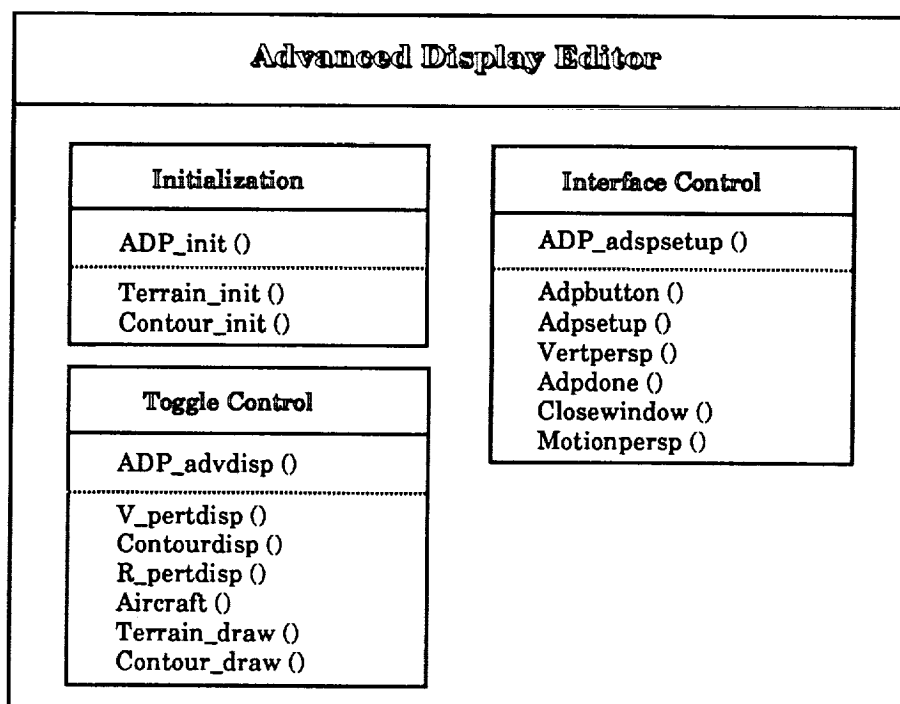


Figure 7. Procedures comprising each Advanced Display function.

4.2.6.6 Parameter Linker : cdelink.c

The Parameter Linker module is constructed by "cdelink.c" (Figure 8), it is used to setup the attributes for the animated graphical object. The animation attributes are referenced by "IDs" instead of absolute coordinates. The major advantage to IDs is that when the user moves the gauge around in the same attached surface, it is unnecessary to relink the animation attributes. The Animation Init utility will convert these IDs to absolute world coordinate numbers to performs the actual 3D transformations. The Parameter Linker can be identified to Interface Control Procedures and Itemize Control Procedures.

4.2.6.6.1 Attribute Linking Control : LNK_linkit

LNK_linkit () is called when "Link Parameter" is selected under the "CDE" menu bar. It controls the main link window and conforms to the final linking procedure.

```
LNK_linkit ( flag )
char flag;      /* = FALSE, create new link */
                /* = TRUE, update old link */
```

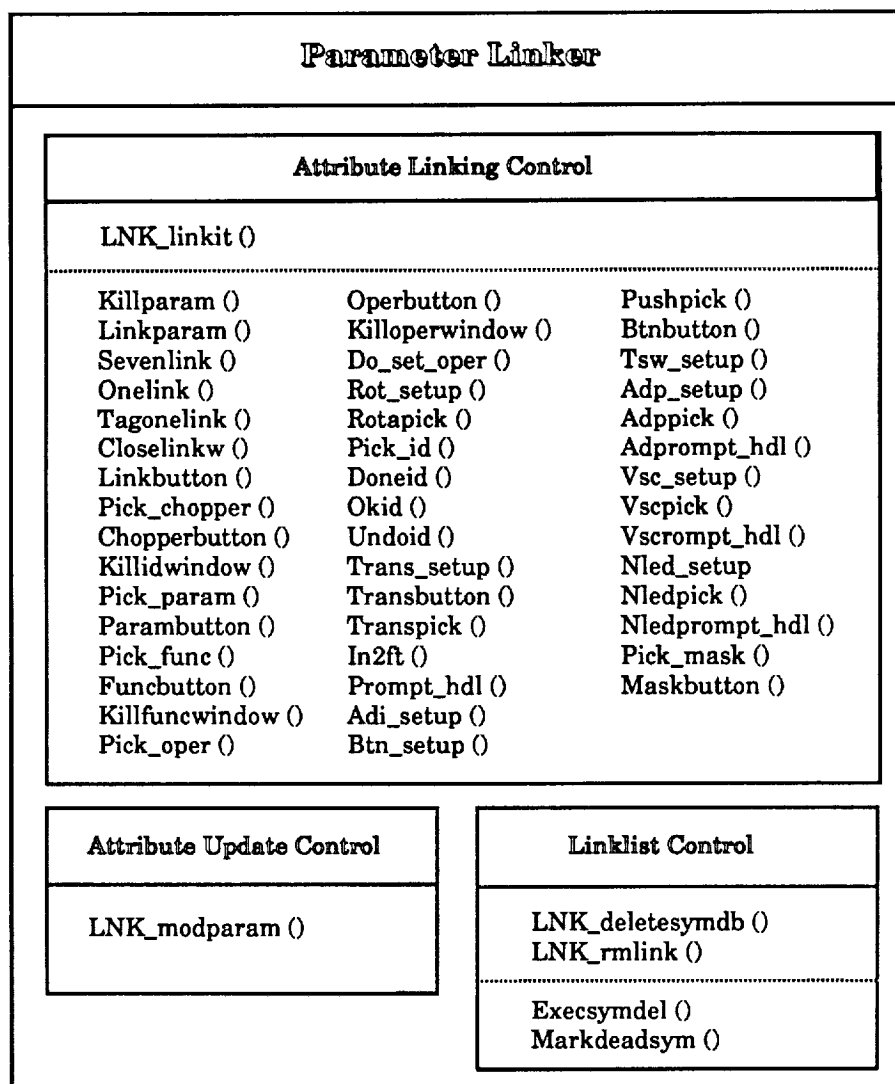


Figure 8. Procedures comprising each Parameter Linker function.

This procedure sets up the animation part of the CDE database structure Dial_infot[]. This Dial_infot[] should cover almost any type of animation. The user should not try to add to or delete any variable in this structure; instead, the meaning for each variable should be reassigned, and the corresponding change made to the function handler(s) in the Animation Control. For changes that can be classified as the Itemize Control (add a model parameter to the menu selection for example), the user should:

1. Declare the new parameter index in the "cdefmt.h".
2. Add the parameter to the menu handler.

3. Modify the corresponding function handler in the Animate Control.
4. Recompile all the programs that relate to the header file "cdefmt.h".

4.2.6.6.2 Attribute Update Setup : LNK_modparam

LNK_modparam () is called when the user selects an instrument for animation attributes updating or examination. This procedure extracts animation attributes from selected instrument and sets up the static variables and calls LNK_linkit for updating interface control.

```
LNK_modparam ( index )
int index;      /* index number for the selected instrument */
```

4.2.6.6.3 Delete a Single Link : LNK_deletesymbd

This procedure allows users to delete a symbd bead and tree below from the CDE database structure by clicking the MultiGen delete icon.

```
LNK_deletesymbd ()
```

Make sure you're in the right mode—A3I bubble is on—before you select the object and execute the delete command, otherwise the program may core dump. More work needs to be put into this command.

4.2.6.6.4 Remove Whole Link : LNK_rmlink

The LNK_rmlink clears all animation arrays and resets all the animation index.

4.2.6.7 Color Handler : cdecolor.c

The Color Handler is designed to enhance color related operations for the CDE. Figure 9 illustrates the functions of the Color Handler.

4.2.6.7.1 Initialization : CDCH_colorinit

The CDCH_colorinit () defines the IRIS color look-up table and sets-up the "writemask" bitplanes.

```
CDCH_colorinit ()
```

The file loaded by CDCH_colorinit is cdecolor.tbl which contains the RGB information for the CDE color table.

4.2.6.7.2 Palette Control : CDCH_palette

The CDCH_palette () is called when the user issues the CDE Color Palette menu command. It displays a palette of available colors and also controls the color palette interface.

```
CDCH_palette ()
```

The palette window handle functions is Cpalredraw and Cpalcontentproc.

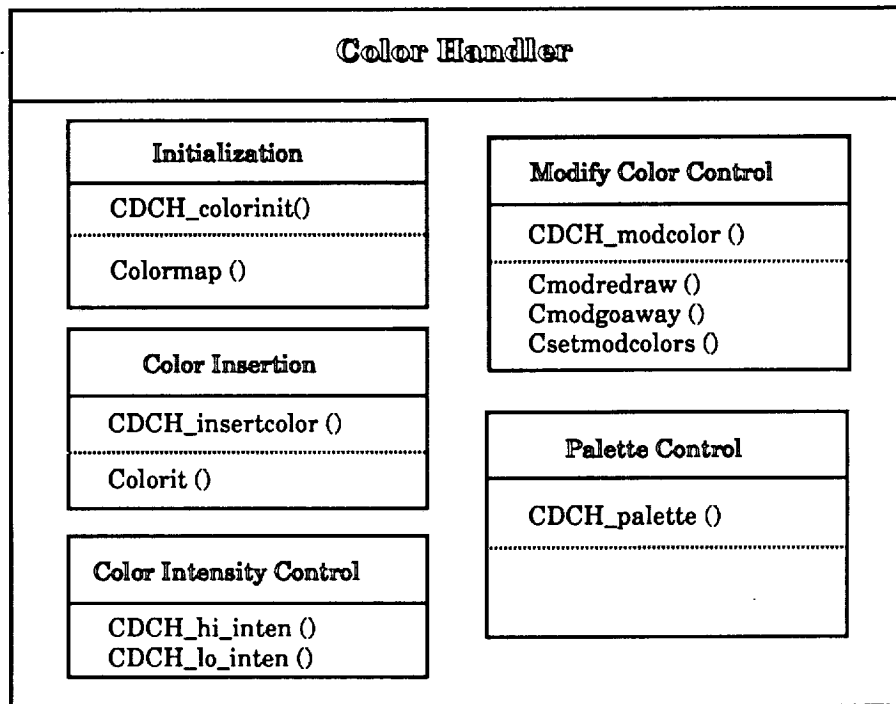


Figure 9. Procedures comprising each Color Handler function.

4.2.6.7.3 Color Insertion : CDCH_insertcolor

This CDCH_insertcolor procedure is called when CDE Insert Color command is issued by the user.

```
CDCH_insertcolor ( function )
int function;          /* function 0: insert current CDE palette color into selected object */
                      /* function 1: change back to original color */
```

4.2.6.7.4 Modify Color Control : CDCH_modcolor

The CDCH_modcolor is called in response to a CDE Modify Color command.

```
CDCH_modcolor ()
```

CDCH_modcolor () allows the user to change the color components of the current color. The window displayed by CDCH_modcolor will have a write flag to indicate if the new color table should be written to disk.

4.2.6.7.5 Color Intensity Control : CDCH_hi_inten, CDCH_lo_inten

CDCH_hi_inten () and CDCH_lo_inten () are called during the animation by the animation control functions.

```
CDCH_hi_inten ( iriscol )
int iriscol;      /* IRIS color index */

CDCH_lo_inten ( iriscol )
int iriscol;      /* IRIS color index */
```

These two procedures allow the user to swap the face colors between the predefined color pair which can simulate the operations such as lighted switch, warning light, etc..

4.2.6.8 Database Manager : cdepage.c

This following three-part document is written for programmers and software designers who want to create or modify the CDE descriptive database (Figure 10). The implementation of the CDE descriptive database is based on MultiGen Data Base Logic and Flight Data Base; familiarity with these two modules as well as the Interface Manager and the MultiGen Kernel are necessary in order to understand this program.

4.2.6.8.0 Database Field Elements Structure

Defined as a local structure, these four structures specify what type of information, editable or noneditable or function pointer, should be displayed on the screen.

```
/* field definition for getfield routines --- defined in mgfmt.h */
typedef struct {
    char type;          /* the data type, INT, FLOAT, etc. */
    char editflag;      /* EDIT, NONEDIT, or READONLY */
} fieldtype, farray [ 1 ];

/* File Header window information */
static fieldtype cdeheadft [ 2 ] = {
    STRING, EDIT,      /* Database filename */
    STRING, NONEDIT    /* First console name */
};

/* Console information window arrangement */
static fieldtype cdeconsoleft [ 8 ] = {
    STRING, EDIT,      /* name */
    STRING, EDIT,      /* description */
    STRING, NONEDIT,   /* MultiGen ID */
    STRING, NONEDIT,   /* pre console */
    STRING, NONEDIT,   /* next console */
    STRING, NONEDIT,   /* centroid x */
    STRING, NONEDIT,   /* centroid y */
    STRING, NONEDIT    /* centroid z */
};
```

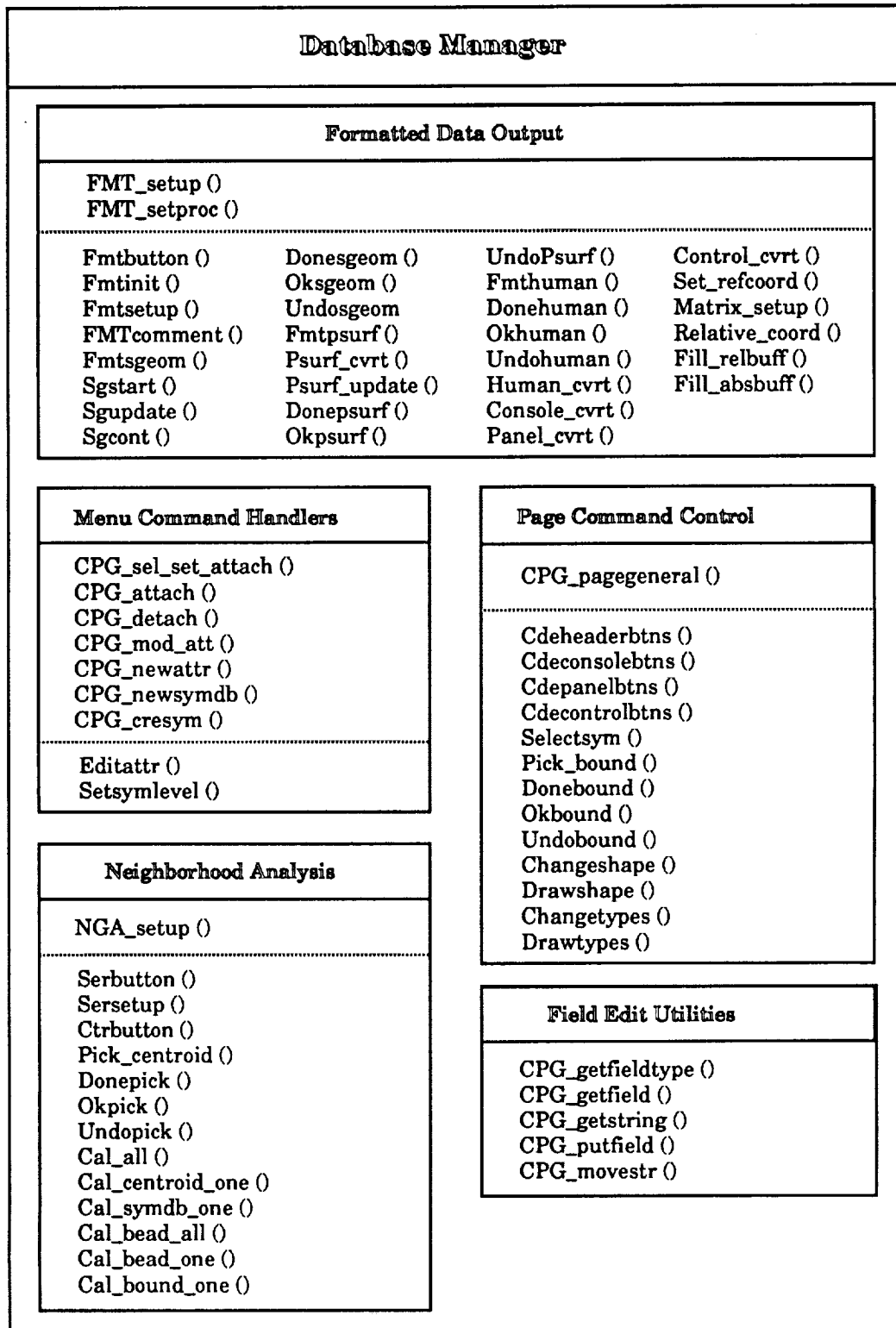


Figure 10. Procedures comprising each Database Management function.

```

/* Panel information window arrangement */
static fieldtype cdepanelft [ 12 ] = {
    STRING, EDIT,          /* name */
    STRING, EDIT,          /* description */
    STRING, NONEDIT,       /* MultiGen ID */
    STRING, NONEDIT,       /* CONSOLE name */
    STRING, NONEDIT,       /* pre panel */
    STRING, NONEDIT,       /* next panel */
    STRING, NONEDIT,       /* centroid x */
    STRING, NONEDIT,       /* centroid y */
    STRING, NONEDIT,       /* centroid z */
    STRING, EDIT,          /* normal height */
    GENERAL, EDIT,         /* shape */
    GENERAL, EDIT,         /* type */
};

/* Control information window arrangement */
static fieldtype cdecontrolft [ 10 ] = {
    STRING, EDIT,          /* name */
    STRING, EDIT,          /* description */
    STRING, NONEDIT,       /* MultiGen ID */
    STRING, NONEDIT,       /* CONSOLE name */
    STRING, NONEDIT,       /* PANEL name */
    STRING, NONEDIT,       /* pre control */
    STRING, NONEDIT,       /* next control */
    STRING, NONEDIT,       /* centroid x */
    STRING, NONEDIT,       /* centroid y */
    STRING, NONEDIT,       /* centroid z */
};

```

4.2.6.8.1 Structure Pointer : CPG_getfieldtype

This function is called when the program tries to locate a selected level of the database structure pointer.

```

CPG_getfieldtype ( level, ft )
int level;          /* symdb level ( header, console, panel, ... etc ) */
fieldtype **ft;     /* return as pointer into fieldtype array */

```

4.2.6.8.2 Field Format Control : CPG_getfield

The CPG_getfield is called when the user wants to know the size of the selected level and field.

```

long CPG_getfield ( symdb, level, field )
dialparampt symdb; /* symdb bead pointer */
int level;          /* database level */
int field;          /* field order in the datafield structure */

```

Because the current database uses string only, that is, only length of string is returned.

4.2.6.8.3 String Field Control : CPG_getstring

This function fetches information from a selected symdb bead and converts it into a defined length of string.

```
CPG_getstring ( symdb, level, field, cnt, string )
dialparampt symdb; /* symdb bead pointer */
int level;          /* database level */
int field;          /* field order in the datafield structure */
int cnt;            /* not used */
char *string        /* converted string pointer */
```

4.2.6.8.4 Store Editable String to Database : CPG_putfield

This function verifies an input field string, if valid, stores the new field value into database.

```
CPG_putfield ( topib, symdb, level, field, val )
ibeadpt topib;      /* not used */
dialparampt symdb; /* symdb bead pointer */
int level;          /* database level */
int field;          /* field order in the datafield structure */
int val;            /* input string */
```

The verification is done by calling CPG_movestr procedure.

4.2.6.8.5 Verify the Editable String : CPG_movestr

This procedure is called when the user wants to make sure the input editable string is a less defined field size; if not, null string returned.

```
CPG_movestr ( to, from, len )
char *to;           /* verified output string */
char *from;         /* input string */
int len;            /* defined string length */
```

4.2.6.8.6 GENERAL Field Handler : CPG_pagegeneral

This procedure is called when the user mouse on the attribute field when GENERAL is defined in the fieldtype structure. Currently, this utility only appears on the panel level, it will rotate a list of shapes or types for the panel.

```
CPG_pagegeneral ( w, delta, size, level, field, editflag )
windowpt w; /* attribute window pointer */
point *delta; /* leftbottom coordinate for the given field */
point *size; /* rightbottom coordinate for the given field */
int level; /* database level */
int field; /* field number in the structure */
int editflag; /* not used */
```

4.2.6.8.7 Modify Attributes Window Control : CPG_mod_att

The CPG_mod_att is called when the user issues the "Modify Attribute" command under "Structure" menu bar, and the "A3I" bubble is turned on.

```
CPG_mod_att ( w )
windowpt w;          /* top window pointer */
```

This procedure will display the CDE descriptive database attribute window for the selected symdb, if nothing is selected, than the header attribute window will be displayed.

4.2.6.8.8 New Attributes Window Control : CPG_newattr

In the previous command, if the selected bead has not yet be defined by the CDE database, than the CPG_newattr will be called to prompt the user on whether a CDE database link for that bead must be created.

```
CPG_newattr ( bead, level )
ibeadpt bead;          /* selected ibead pointer */
int level;             /* database level */
```

4.2.6.8.9 Symdb Structure Init : CPG_newsymdb

This procedure will return a new and initialized symdb database bead.

```
dialparampt CPG_newsymdb ()
```

4.2.6.8.10 Create New Link to Symdb Structure Tree : CPG_cresym

The CPG_cresym is called when the user wants to create a link for a given symdb.

```
CPG_cresym ( symdb, level, uplevel, superbead )
dialparampt symdb;          /* given symdb pointer, can be NULL */
int level;                  /* database level */
dialparampt uplevel;        /* parent symdb pointer, can be NULL */
dialparampt superbead;      /* super bead pointer */
```

4.2.6.8.11 Attach Link to Parent Symdb Structure Tree : CPG_attach

This procedure is called when the user issues the "Attach" command under the "Structure" menu bar with the "A3I" bubble on.

```
CPG_attach ( sbead )
dialparampt sbead;          /* given child symdb pointer */
```

This procedure will first verify the attached database level and parent bead, then clear the old database link for the given symdb, and create a new link for the given symdb.

4.2.6.8.12 Detach Link from Parent Symdb Structure Tree : CPG_detach

The CPG_detach is called when user clicks on the "Structure" menu "Detach" command with the "A3I" bubble on. This procedure will unlink the selected symdb from its parent and sibling.

CPG_detach ()

4.2.6.8.13 Set Parent Symdb : CPG_sel_set_attach

The CPG_detach is called when user clicks on the "Structure" menu "Set Parent" command with "A3I" bubble on. This procedure will set the first selected symdb as parent bead.

CPG_sel_set_attach ()

4.2.6.9 Database Manager : cdefmter.c

The purpose for this file is to output different database formats for other applications. Currently, S-geometry, PSURF, and CDE descriptive database three type of databases can be output from MultiGen environment.

4.2.6.9.1 Selection Window : FMT_setup

This procedure is called when the user issues a CDE "data format O/P" command. It displays a selection window of available data formats and prompts the user for selection input.

FMT_setup ()

4.2.6.9.2 Control Window Processes : FMT_setproc

This procedure will setup the function handlers that response to the UNDO, OK, and DONE commands in the control window.

```
FMT_setproc ( donefunc, okfunc, undofunc, msg )
int ( *undofunc ) (); /* called when UNDO button invoked */
int ( *okfunc ) (); /* called when OK button invoked */
int ( *donefunc ) (); /* called when DONE button invoked */
int msg; /* message ID */
```

4.2.6.10 Database Manager : cdeneig.c

This program module performs neighborhood analysis for the CDE descriptive database. NGA_setup is called when the user clicks on the "Neighborhood" button in the attribute window. This will cause a window to be displayed and prompt the user for input selection. After the user defines all the variables, the program will compute the distance between the given symdb and every symdb that exists in the same datalevel with the given symdb. For the symdb's where the distances are smaller than the defined radius, their names will be displayed on the viewing window.

5.0 Notes

5.1 Miscellaneous

The purpose of this section is to include any additional information which would be beneficial in understanding your CSCIs design, implementation, or operation. This could be notes from

meetings or conferences, results of testing, and any unique features in the software that have been included to streamline future enhancements, such as special modularity "hooks".

5.2 Limitations

Discuss any limitations of your CSCI, as well as any requirements which we not met during this phase.

5.3 Future Directions

Describe any future enhancements or modifications which should be made to your CSCI. Use you experience, "lessons learned", and domain knowledge to suggest what could be pursued during further development.

6.0 Users Guide

The purpose of the user's guide is to provide end users (cockpit designers) with the necessary information to effectively operate the CDE.

6.1 Introduction

It would not be advisable for people to work on the CDE without prior knowledge of MultiGen. Getting the most out of the CDE isn't easy. To benefit from this document, you should have a basic understanding of MultiGen. You should know how to use MultiGen to create, view, and modify objects.

6.2 Related Documentation

First time users of MultiGen or the CDE should familiarize themselves with the *MultiGen Modeller's User's Guide* before attempting to use the system. The detailed instructions for using the CDE utilities are listed in this chapter.

6.3 The Structure of CDE Hierarchical Database

Both the MultiGen and CDE databases are hierarchical. The hierarchical database system is based on records that are organized into inverted tree structures. There is a single "root" record, with sub-records associated with the root. Each sub-record has its sub-records, and so on. All sub-records on the same level have the same format, but formats can differ from level to level. Each particular "parent" record may have many "child" records, but a child record has only one parent record. Figure 11 shows a small sample from CDE hierarchical database.

The dotted line represents the sibling relationship. Currently, all objects animated on the CDE are limited to the Control Level. Some instruments contain there own hierarchical structure. For example, the HSI, one cannot compute the final location of the Course Deviation Bar without first computing the Azimuth Indicator. In this case, when the designer creates the graphical object of HSI, he should make sure the Course Deviation Bar is the child or subbead of the Azimuth Indicator. During the CDE animation, the children transformation are relative (instead of independent) to their parent. Generally speaking, it is always a good practice to construct each

instrument/gauge in MultiGen's "Cluster" level and each "movement" (animated object) in the "object" level. For some types of gauges which have more than three hierarchical levels, like HSI, use the MultiGen "Subface" to generate the extra parent-child relationship.

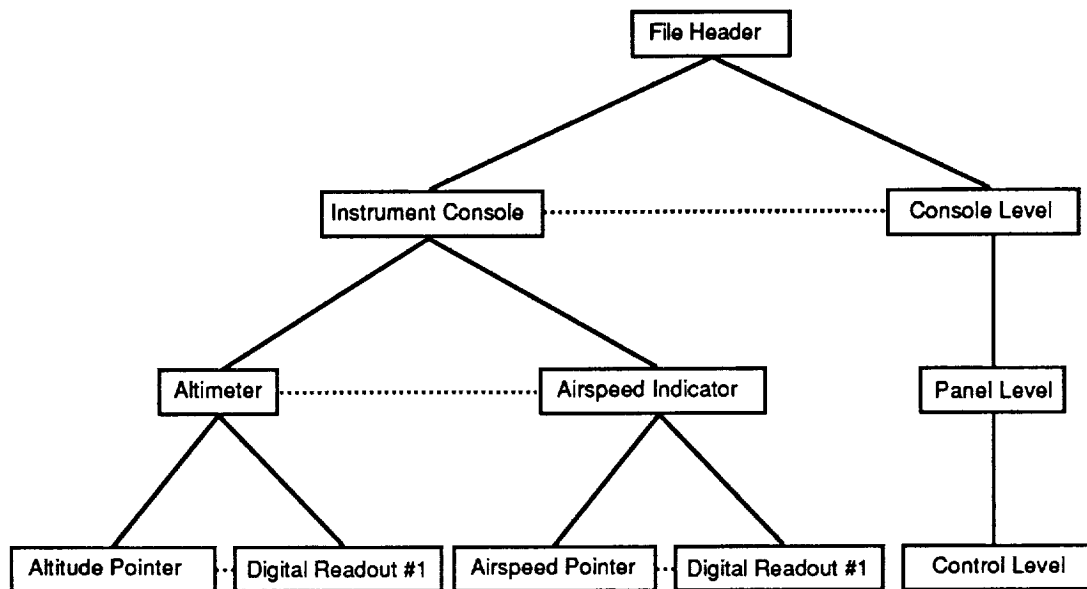


Figure 11. Sample structure for CDE hierarchical database.

6.4 CDE Menu Commands

As shown in Figure 12, the CDE has fourteen pull-down menu commands.

6.4.1 Open Coord. File

When the user issues this command the program brings up a temporary window that contains the name of the last requested file in highlighted text. Press RETURN to confirm, or edit the filename then press RETURN. If the requested file exists, the data will be converted to the MultiGen i-format and the object will be drawn on the active window. If necessary, you can edit the extracted graphic object by using the MultiGen commands. When you are satisfied with your results, use the MultiGen copy and paste commands to store the new object in the database.

6.4.2 New Gauge

New Gauge command pops up a menu that contains a list of the predefined instruments. Select the desired instrument by clicking on the bubble then clicking the "OK" button. For each selection, a pop-up menu appears on the screen. Type in the appropriate information using the normal text editing conventions. After the user clicks on the "SHOW" button, the edit control window appears on the upper-left of MultiGen desktop. This control window provides

instructions on the gauge paste procedures. The first instruction is to select the gauge/instrument center. Choose the center by pointing the cursor to the desired location and clicking the left mouse button. The edit control window then prompts for the zero alignment point (normal rest position for the pointer), depress the left mouse button and drag the dotted reference line to align with the pointer, then release the mouse button. After you click the "done" button in the edit control window and a construction instrument will be pasted on the active database window. Click "UNDO" to delete the pasted instrument or click the "DONE" button to store it in the database.

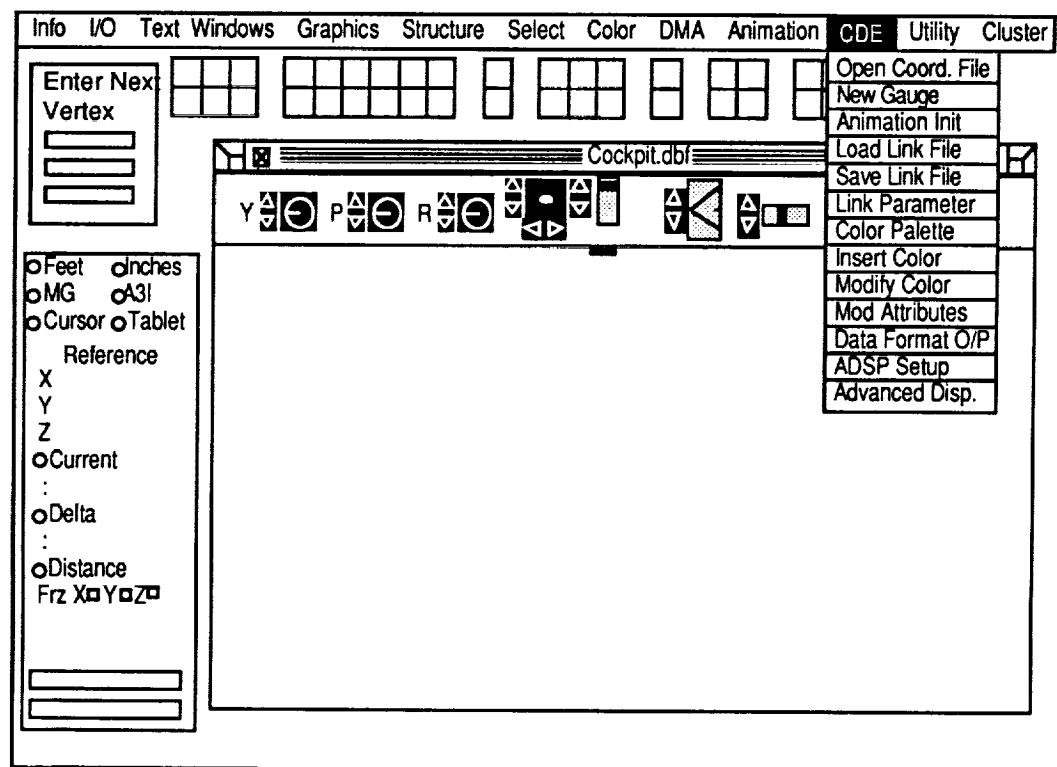


Figure 12. CDE pull-down menu commands.

6.4.3 Animation Init

Animate Init computes the 3D animation data (i.e. transformation matrix) for each instrument in the link list which is created by *Load Link File* and/or by *Link Parameter* commands. Whenever the link list or the instrument physical layout has been altered, the user has to execute this command to set-up the new transformation matrices.

6.4.4 Load Link File

Load Link File brings up a temporary window that contains the default filename. Press RETURN to confirm, or enter the new filename then press RETURN. This command loads the

linkfile from the disk and then appends it to the CDE parameter link list. After you load a new linkfile you have to execute *Animate Init* to update the 3D animation data.

6.4.5 Save Link File

Save Link File brings up a temporary window that contains the name of the last requested linkfile as selected text. Press RETURN to confirm, or edit the filename and then press RETURN. This command will save the current CDE parameter link list to disk.

6.4.6 Link Parameter

6.4.6.1 Chopper

The user can use this function to select one of the helicopters from the list to be the target. CDE will fetch the simulation system/aero dynamic data from this helicopter.

6.4.6.2 Parameter

A list of available system/aero dynamic parameters is provided by the CDE. You can pick one of the parameter as the input variable for the instrument.

6.4.6.3 Function Handler

Currently seven mapping functions are available for the user.

6.4.6.3.1 Linear Mapping Function

$$F(X) = A * X + C \quad A, C \text{ are constants}$$

This is the most commonly used function. Output is directly proportional to the input weighted by a scale factor "A" and an offset factor "C". The mathematical relationship of this function is shown below.

A	C	X	F (X)
0.01	0	14964.2	149.642
0.1	0	14964.2	1496.42
1	0	14964.2	14964.2
10	0	14964.2	149642.
100	0	14964.2	1496420.

6.4.6.3.2 Periodic Mapping Function

$$F(X) = C * (X - A * ((\text{int})(X/A))) \quad A, C \text{ are constants.}$$

This function handle those inputs which function value recurs at a regular interval, such as heading angle. The period (repeat interval) is defined by "A" and the output is scaled by "C". The mathematical relationship of this function is shown below.

A*	C*	X	F (X)
120	1	14964.2	84.2

200	1	14964.2	164.2
360	1	14964.2	204.2
400	1	14964.2	164.2
480	1	14964.2	84.2
600	1	14964.2	564.2

*note: in this function C is the scale factor and A is the period.

6.4.6.3.3 Truncate Mapping Function

$$F(X) = (\text{int})(A * X) + C \quad A, C \text{ are constants.}$$

Similar to the Linear Function, but the fraction part of scaled value is discarded. The output is the integer portion of the scaled value plus the offset "C". The mathematical relationship of this function is shown below.

A	C	X	F(X)
1	0	14964.2	14964
10	0	14964.2	1496
100	0	14964.2	149
1000	0	14964.2	14
10000	0	14964.2	1
100000	0	14964.2	0

6.4.6.3.4 Linear Digit Mapping Function

$$F(X) = (X/A - (\text{int})(X/A)) * 10 + C \quad A, C \text{ are constants.}$$

This function was specially designed for the last digit (linear digit) of Drum Digit display. It works like a reversed truncate function; it removes digits located on the left of a specific digit for a given number value. The constant "A" decides which digit is the reference digit, after the conversion, the reference digit will be placed on the ones position.

A	C	X	F(X)
1	0	14964.2	2
10	0	14964.2	4.2
100	0	14964.2	6.42
1000	0	14964.2	9.642
10000	0	14964.2	4.9642
100000	0	14964.2	1.49642

6.4.6.3.5 Drum Digit Mapping Function

$$\text{TMP1}(X) = (X/A - (\text{int})(X/A)) * 10 + C \quad A, C \text{ are constants.}$$

$$\text{TMP2}(X) = (\text{int}) \text{TMP1}(X)$$

$$\text{TMP} = \text{TMP1}(X) - \text{TMP2}(X)$$

$$F(X) = \begin{cases} \text{TMP2}(X) + 10 * (\text{TMP}(X) - 0.9) & \text{if } \text{TMP}(X) > 0.9 \\ \text{TMP2}(X) + 10 * (\text{TMP}(X) + 0.9) & \text{if } \text{TMP}(X) < -0.9 \end{cases}$$

TMP2 (X)

if $-0.9 \leq \text{TMP} (X) \leq 0.9$

This function is specially designed for the Drum Digit display. It is possible to extract specific digit(s) of a given number value. The output value depends on the digit to its right, if that digit value is between 0 and 9, then the function returns a single digit, otherwise, a sum of a handover fraction and that single digit is returned.

A	C	X	F (X)
1	0	14964.2	2
10	0	14964.2	4
100	0	14964.2	6
1000	0	14964.2	9
10000	0	14964.2	4.642*
10000	0	18956.23	8.5623*
100000	0	14964.2	1

*note: the handover fraction is composed by the digits on the right-side of digit "9".

6.4.6.3.7 Natural Logarithmic Function

$$F (X) = A * \ln (X) + C \quad A, C \text{ are constants.}$$

This natural logarithmic function allows user to view a great range of data in a reasonably small scale. The value for X should be positive, however, the CDE will automatically switch to the linear function if X becomes negative.

A	C	X	F (X)
1	0	-1.4964	-1.4964
1	0	1.4964	0.4031
1	0	14.9642	2.7057
1	0	149.642	5.0082
1	0	1496.42	7.3108
1	0	14964.2	9.6134

6.4.6.3.8 User Defined Function

6.4.6.4 Operation

The *Operation* and *Function Handler* are related to each other. The *Function Handler* tunes the input parameter to the Device Normal Data (DND). It is the DND, not the input parameter, that drives the *Operation*. It's important that the user clearly understands the output of the *Function Handler* before using this operation. Currently, eight operations are available for the animation purposes.

6.4.6.4.1 Rotation

This function can be used to perform any rotation of a given object. The menu definitions are given below.

Starting Angle:	Zero degree means the original pointer position is the starting position, any positive X value will cause the pointer to rotate clockwise X degrees to the starting position.
Sweep Area:	This is the range. This value should represent the number of degrees you expect the pointer to rotate "clockwise" when input DND reaches the Maximum Limit.
Maximum Limit:	This is the value when pointer at the "Starting Angle + Sweep Area" location.
Minimum Limit:	This is the value of the pointer at the starting angle (initial location).
Rotation Center:	ID for the Center point of the rotation.
Movement ID:	The graphic ID for the object you want to rotate.
Reference Face ID:	The surface where you perform the rotation.

6.4.6.4.2 Translation

You can use this function to perform any translation for a given object. The definitions for the menu items are listed below.

Down +--+ Up:	Click this bubble if you want the object to move upward when the DND increases.
Left +--+ Right:	Click this bubble if you want the object to move to the right when the DND increases.
Displacement:	The reference interval for the range.
Corresponding rang:	The DND value that cause the object to travel the length of previous defined Displacement.
Instrument Center ID:	This can be any point, except those two points in the Horiz. Alignment Axis, that reside in the same plane with the Movement.
Horiz. Alignment Axis:	This axis sets up the horizontal reference axis. Because the nature of the graphics implementation, always pick the lower line as axis.
Movement ID:	The graphic ID for the object you want to rotate.
Reference Face ID:	The surface where you perform the rotation.

6.4.6.4.3 ADI

Attitude Display Indicator (ADI): This operation is designed for ADI only.

Pitch Ladder Displacement: The physical size of the ADI's pitch ladder.

Corresponding Pitch Angle: This is the pitch angle range for the previous defined Pitch Ladder Displacement.

Rotation Center: ID for the Center point of the rotation.

Horiz. Alignment Axis: This axis sets up the horizontal reference axis. Because of the nature of the graphics implementation, always picks the lower line as axis.

Movement ID: The graphic ID for the object you want to rotate.

Reference Face ID: The surface where you perform the rotation.

6.4.6.4.4 Pushbutton

This function simulates a lighted pushbutton, it also can be used to simulate warning lights.

Pushbutton FACE ID: The graphic ID for the top surface of a pushbutton or the cover of the warning light. Make sure the color for that face belongs to the CDE color pair.

Switch Value: The critical DND value will change the state.

High intensiABOVE..: When the DND value is above the Switch Value, the upper color of that color pair will be displayed. Otherwise, it will display the lower color.

High intensiBELOW..: When the DND value is below the Switch Value, the upper color of that color pair will be displayed. Otherwise, it will displayed the lower color.

6.4.6.4.5 Toggle Switch

A two or three state toggle switch can be simulated by this function.

Flip Up Value: When the DND value is above this setting, the toggle switch will flip up.

Flip Down Value: When the DND value is below this setting, the toggle switch will flip down. If the Flip Up and Flip Down values are the same, than it is a two state toggle switch, otherwise it is a three state toggle switch. When the value of a three stage toggle switch is between these two settings, its handle will remain the same.

Instrument Center ID: ID for the Center point in the base of the toggle switch.

Alignment Axis: This axis sets up the horizontal reference line. Because of the nature of the graphics implementation, always picks the lower line as axis.

Movement ID: The graphic ID for the handle of the toggle switch.

Attached Face ID: The base of the toggle switch.

6.4.6.4.6 Advanced Disp

This operation will define the window location for the Advanced Display.

Attached Face ID: The rectangular window Face ID.

Lower Left Vertex ID: The point locates in the lower left of the window.

Upper Right Vertex ID: The point locates on the upper right of the window.

6.4.6.4.7 Vertical Scale

For this function the vertical scale has to be in full scale position when created.

Maximum Limit: This is the value when the scale reaches the maximum mark.

Minimum Limit: This is the value when scale reaches the minimum mark.

Scale ID: The ID for the scale movement.

Upper Left Vertex ID: The point locates in the upper left of the scale movement.

Upper Right Vertex ID: The point locates on the upper right of the scale movement.

6.4.6.4.8 Numerical LED

CDE's LED Operation always displays the "ones" digit within the DND number value. If you want to display the "hundredth" digit for the input parameter, define the function handler as the Linear Function with $A=0.01$, $C=0$, this will make the DND's "ones" place equal to the input parameter's "hundredth" place giving you the desired results. Make sure to follow the segment order illustrated in the menu window.

6.4.6.4 Writemask

The Writemask utility controls the CDE's writemask flag for this object.

6.4.7 Color Palette

Color Palette opens a window containing the CDE color palette. To select a color, position the cursor to the desired color then click the left mouse button. The selected color will appear in the current color block at the left of the palette, along with the color number. The middle mouse button controls the vertical movement of the cursor, the right mouse button controls the horizontal movement. This palette includes Erasable colors, Unerasable colors, and Unerasable color pairs. Normally, there is no difference between the Erasable and the Unerasable colors. When the CDE's writemask flag is TRUE, however, the Unerasable colors can cover-up the Erasable colors. With this property in mind the user can perform some "windowing" effect animations such as ADI and Drum Dial gauges.

6.4.8 Insert Color

This command changes the color of the selected object(s) with the newly selected color.

6.4.9 Modify Color

Modify Color brings up a window containing the current color box and three color bars. Drag the tabs in the red, green, or blue color bars, and the change will be reflected in the current color box, color palette, and database window. If the check mark is on in the "Write File" box, the changed color palette will be written to disk when the color modification window is closed.

6.4.10 Mod Attribute

Mod Attributes allows the user to browse through the CDE link list. The viewing window contains the data base ID and title of every instrument in the link list. To view or modify the link list, first select the desired item in the viewing window and click on the "Selected Instrument" box. The rest of the procedures are the same as the *Link Parameter*.

6.4.11 Data Format O/P

6.4.12 ADSP Setup

ADSP Setup brings up a control window on the screen, which is used to control the view point for the Vertical-type Perspective Display. The user can change the viewing angles, viewing distance, prediction time, and scaling factor. The changes are reflected in the Advanced Display window.

6.4.13 Advanced Disp

Advanced Display toggles the scene in the Advanced Display window from No-view to Vertical-type Perspective Display to Motion Perspective Display to Radar Display.

6.4.14 Unlink All

Unlink All removes all the symdb links, clears all the animation arrays and resets all the animation indexes.

6.5 Error Messages and Diagnostics

The CDE software sounds a bell when an error occurs. The user can check the cause of this error by executing the "Last Error Msg" command under the Info Menu. "Last Error Msg" displays the last error message. In addition, the Status Window command under the Info Menu can open a window that displays a log of events that have occurred during the current edit session.

Annex D

Army-NASA Aircrew/Aircraft Integration Program

A³I

**Software Detailed Design Document:
Anthropometric Manikin Model "Jack"**

prepared by

Gretchen Helms

December 1988

Table of Contents

1.0 INTRODUCTION.....	D-1
1.1 Identification.....	D-1
1.2 Scope.....	D-1
1.3 Purpose	D-1
2.0 RELATED DOCUMENTATION	D-2
2.1 Applicable Documents.....	D-3
2.2 Information Documents	D-3
3.0 REQUIREMENTS AND DESIGN APPROACH	D-3
3.1 Requirements and Rationale	D-3
3.2 Hardware Environment	D-5
3.3 Software Environment	D-5
4.0 DETAILED DESIGN DESCRIPTION	D-5
4.1 Organization.....	D-5
4.2 Unit Detailed Design.....	D-5
4.2.1 Demo Directory.....	D-5
4.2.2 Psurf Files.....	D-6
4.2.3 Environment Files.....	D-7
4.2.4 Animation Files.....	D-8
5.0 NOTES.....	D-9
5.1 Limitations.....	D-9
5.2 Future Directions.....	D-9
6.0 USERS GUIDE.....	D-10
6.1 Compiling Jack.....	D-10
6.2 Porting Jack	D-12
6.3 Modes of Operation.....	D-12
6.4 Updating Jack	D-13
6.5 Creating Animation Scripts	D-14
6.6 Generating Movements Automatically.....	D-16
6.7 Downloading from CDE.....	D-16
6.8 Jack Installation Instructions	D-16
7.0 APPENDICES	D-19
A. Jack Directory Detailed Breakdown	D-19
B. Sample Environment File: radiosites.env	D-24
C. Jack Demo Instructions	D-29
D. "Makescript.c", Movement Generating Program	D-32

1.0 INTRODUCTION

1.1 Identification

This document establishes the requirements and detailed design of the Jack Computer Software Configuration Item (CSCI), which forms a part of the A3I Computer Program System. Descriptions of the detailed processing requirements, structure, I/O, and control are provided for each lower level Computer Software Component (CSC), unit, or function contained within the CSCI.

1.2 Scope

This document describes the function, composition and use of the Jack software used in Phase III of the A3I simulation. Since Jack has its own set of detailed user's and programmer's manuals published by UPenn, this document will complement such data and describe the files created for A3I, along with instructions for porting, recompiling and updating Jack. The document assumes that the reader is familiar with the UNIX operating system, the C programming language, and animation techniques.

1.3 Purpose

Jack, the software described within this document, is used to observe how a human mannequin interacts with its environment and what effects body types will have upon performance of a task. Jack comes with a variety of different body types pre-defined and known to the system, both male and female bodies, ranging from the 5th to 95th percentile. Each mannequin is fully articulated and manipulative and reflects the joint limitations of a normal human. Jack is used to generate the wireframe and solid displays of a pilot with a cockpit created using the A3I Cockpit Design Editor and performing reaches. Jack was developed at the University of Pennsylvania under the direction of Dr. Norman Badler, while implementation for the A3I Project was done by Gretchen Helms. Jack currently runs on the IRIS 4D70 Turbo graphics workstation running 4Sight and System 3.0, distributed by Silicon Graphics. The software is written in C, with slight modifications that provide for cross-machine communication with the Symbolics 3675.

As indicated in Figure 1, the current Jack software interacts with a number of A3I's computers, files, and system programs, as well as the user. The primary purposes of Jack are: 1) to create or modify an environment in which a mannequin may be placed, and 2) to observe the interaction of the mannequin with the environment. Environments are not limited to that of a helicopter cockpit, nor is the mannequin limited to the cockpit. Any environment can be created, and any number of mannequins can be placed anywhere in that environment.

To aid in analyzing how well the mannequin works with its environment, two tools are particularly useful:

- Animation Facilities

Jack allows the user to create a file which contains a sequence of commands that Jack executes. By commanding Jack to execute body movements, the movements of the pilot in the

cockpit can be duplicated. Furthermore, by attaching the view of the Jack environment to the eye of the mannequin before executing the animation script, the program displays an environment perspective corresponding to what the mannequin would see while moving in the cockpit, allowing conclusions to be made about cockpit occlusion and visibility.

- **Reach Facilities**

This facility causes the mannequin to move a specified portion of its anatomy to a pre-chosen point in space, reporting back the final distance from the goal site. It is necessary to pre-pick the sites the mannequin will move to before starting a reach. The success or failure of the mannequin to reach a site helps in analyzing problems with the pilot's size and other constraints.

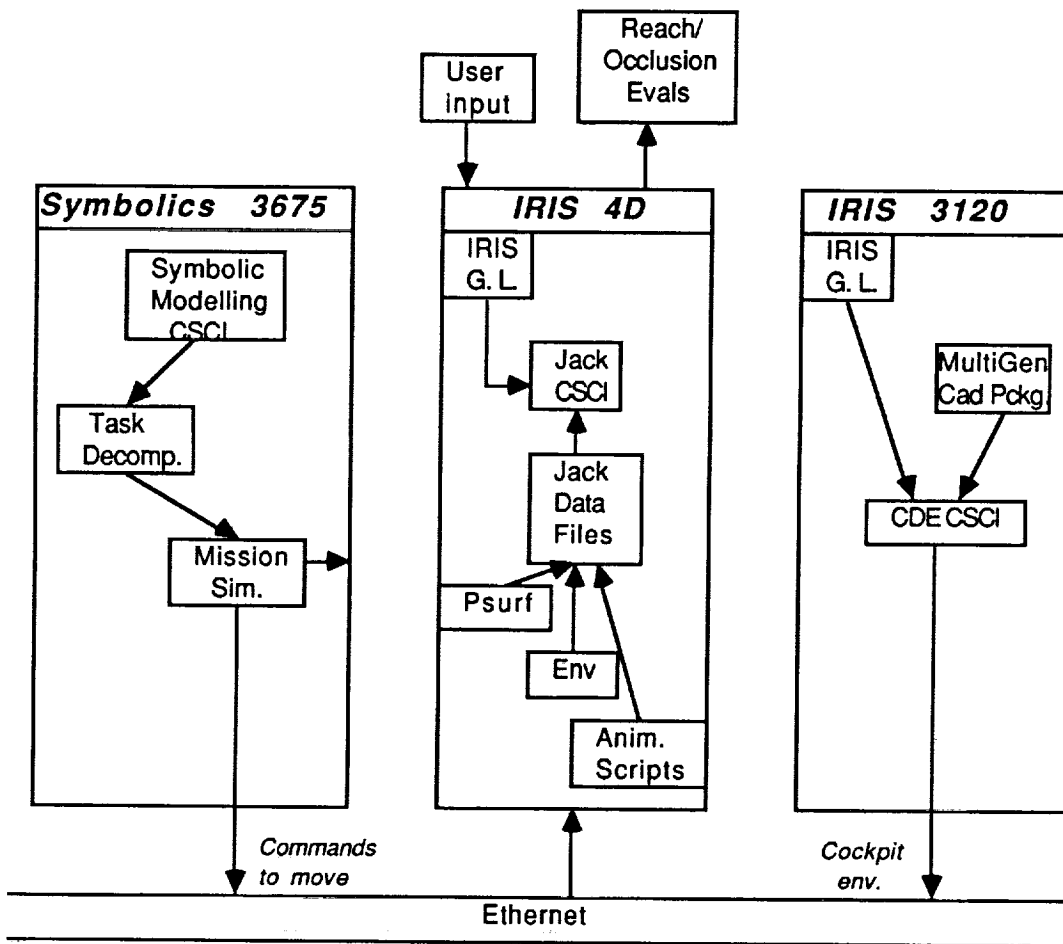


Figure 1. Jack/A3I Workstation Interaction

2.0 RELATED DOCUMENTATION

2.1 Applicable Documents

Silicon Graphics Inc., *IRIS User's Guide*, Volume I and II, Version 3.0, Mountain View, California, 1986.

Cary B. Phillips, *Jack User's Guide*, Version 2.0, Computer Graphics Laboratory, Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia, Pennsylvania 19104-6389, September 12, 1988.

Cary B. Phillips, *Programming with Jack*, Second Edition, Jack Version 3, Computer Graphics Laboratory, Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia, Pennsylvania 19104-6389, June 9, 1988,

Stephen G. Kochan, *Programming in C*, Hayden Books, 4300 West 62nd St, Indianapolis, Indiana 46268, 1988

Teh-Ming Hsieh, *A3I Phase III CDE Software*, A3I Project, NASA Ames Research Center, February 1989.

Huimin Chiu, *A3I Phase III Flight Dynamics Guidance*, A3I Project, NASA Ames Research Center, February 1989 .

2.2 Information Documents

Simulating Personnel and Tasks in a 3-D Environment, Progress Report NO. 32, University of Pennsylvania, Department of Computer and Information Science, School of Engineering and Applied Science, Philadelphia, PA 19104-6389, September 20, 1988

Norman I. Badler, *A Representation for Natural Human Movement*, MS-CIS-86-23, Graphics Lab 13, University of Pennsylvania, Department of Computer and Information Science, School of Engineering and Applied Science, Philadelphia, PA 19104-6389

Dr. Norman Badler, *Modeling and Animating Human Figures in a CAD Environment*, MS-CIS-86-88, Graphics Lab 14, University of Pennsylvania, Department of Computer and Information Science, School of Engineering and Applied Science, Philadelphia, PA 19104-6389

Jeffrey Esakov and Norman I. Badler, *An Investigation of Language Input and Performance Timing for Task Animation*, MS-CIS-88-87, Graphics Lab 25, University of Pennsylvania, Department of Computer and Information Science, School of Engineering and Applied Science, Philadelphia, PA 19104-6389

3.0 REQUIREMENTS AND DESIGN APPROACH

3.1 Requirements and Rationale

The Jack CSCI fulfills a requirement for a package allowing graphic three dimensional representation of a human's interaction within his environment. Jack represents the most advanced human modelling system available that can perform the tasks required by our workstations. At the time A3I came into existence, Jack was substantially underway and was an obvious choice for A3I.

The Jack CSCI was developed and used with at least these goals in mind:

- To create and/or alter 3D geometric models of environments within which a mannequin would interact.
- To provide a means of looking at the interaction between the mannequin and the environment the mannequin is in.
- To provide helicopter cockpit designer(s) with a dynamic, visual tool for evaluating the pilot's performance in terms of reach, fit, view and timing under a variety of conditions.

The requirements needed for the model to meet the above stated objectives are:

- A graphical anthropometric representation of the human pilot which can exhibit human-like movement of body segments(s). The movement should reflect constraints imposed on the pilot by flight gear, helicopter vibration, force in flight, etc.
- A graphical representation of the helicopter cockpit with which the animated figure can interact.
- The ability to generate a full range of body types, sizes and masses, each of which can be tested for reach and fit.
- A system which can be integrated into the A3I System in terms of both hardware and software compatibility.
- A dynamic model for describing pilot movement so that the tasks used to describe a helicopter mission in an A3I simulation can also serve as input to this pilot model. The created animation should playback proportionally to the time allotted for the physical task.
- A method to define both function and position of each item on the control panel and to relay these descriptions to the animation being created.
- Adequate documentation and software systems configuration which makes the addition of new features, needed modifications or application dependent changes possible.

The aspects of the IRIS 4D70GT important to Phase III development were: a fast graphics engine in an inexpensive workstation; an extensive library to exercise the engine; a UNIX program development environment, with tools that are widely used and that therefore require little learning for those who have used a UNIX system before; and a relatively powerful CPU for the price (a RISC-based processor with floating-point accelerator).

3.2 Hardware Environment

The IRIS 4DGT as configured for this project contains a RISC-based central processing unit, a floating-point accelerator for the CPU, 12-megabyte program memory, a frame buffer or image memory of 1024x1024x32 bits, a geometry engine for 3D coordinate transformations, a proprietary microcoded display processor and frame-buffer controller, a 19-inch 60 Hz non-interlaced 1280x1024 resolution RGB color monitor, an Ethernet interface with TCP/IP and NFS softwares, a keyboard and a mouse for user input, and two 440-megabyte disk drives.

Of the above mentioned hardware, the most important one is the combination of the geometry engine and display controller in the graphics pipeline which provides high speed rendering. Several other elements are also important. The Ethernet controller and A3I Communications software make possible the interfacing of a Symbolics 3675 and the IRIS without special purpose hardware. The integration of the IRIS graphics library, GL2, with input devices, including a mouse, a button box and a dials box, make it possible to use these devices easily by calling library functions only. Finally, the large disk drives have proved useful for storing sizable quantities of graphics data and code.

3.3 Software Environment

The most important elements of the IRIS 4D70GT software environment are :

- A general-purpose, easy-to-use graphics library, the IRIS Graphics Library II (GL2).
- A standard Ethernet interface protocol (TCP/IP).
- The UNIX V.3 operating system with Berkeley 4.3 and Silicon Graphics enhancements, a C compiler, a development/debugging environment integrated with the C compiler and several hundred UNIX tools.

4.0 DETAILED DESIGN DESCRIPTION

4.1 Organization

4.2 Unit Detailed Design

The following section details each file or directory created by A3I personnel for use with Jack, or changes to the Jack source code that have been made by A3I.

4.2.1 Demo Directory

The /usr/local/upenn/demo directory serves as the space where demo and work files are developed and stored. A file called Index exists, detailing each file in the directory and what its purpose or contents are. It is strictly up to the user to edit and update the Index file.

Each file in the demo directory has a specific suffix attached to it. The meanings of the suffixes are as follows:

- .pss This is a psurf file, the most basic of the files Jack uses. Each line in a psurf file contains information about nodes, lines, edges, faces, and color attributes. This file should only be altered by changing the color attribute of a face. Typically, a psurf file has its origins in vehicles created with the CDE CSCI, which is described in detail by the document referenced in Section 2.1. Psurf files are used by environment files.
- .env This is an environment file. Each line in an environment file contains information about an environment in Jack that the user has created, including position information, how many figures are present, what colors belong to what attribute, and what the figures are rooted to. This file may be altered to change colors, positions, figures, etc. Environment files are created when the user wants to save an environment. Environment files are used by animation files and by reach demonstrations.
- .jcl This is an animation file. Each line in an animation file is a command to Jack, which Jack executes. This file may be altered to reorder, add or delete commands to Jack. Animation files are created by the user to provide a means of moving the figures in an environment, and the commands may be created either while still in Jack, by using a text editor, or by using the animation generator.

Further information about how psurf, environment and jcl files work can be found in the Jack User's Manual.

4.2.2 Psurf Files

The following files in the /usr/local/upenn/demo directory are psurf files:

- ah64.pss This psurf file was downloaded from the Apache model in the CDE system, and contains the full cockpit plus the Apache canopy. It was originally intended for use with the occlusion and reach analysis demos, but it was discovered that the seat wasn't quite positioned properly, necessitating revisions on the CDE cockpit. This version of the Apache cockpit was retained for use by the animation demo, and is referenced by the files view.env, sites.env, scene.env, ah64.env, 95pilot.env.
- cockpit.pss This psurf file is a modified version of the ah64 psurf. The canopy has been eliminated to save time and neaten up the screen, the seat has been moved forward slightly, rudder pedals have been added, and more detail has been added to the radio panel for the reaches. This version of the Apache cockpit is used in the reach demo, and is referenced by the files radiosolid.env, radiosites.env, and cockpit.env.

Jack has certain limitations with psurf files. For instance, the Apache cockpits that we are currently using are based on the 3 dimensional models created with the CDE system. Unfortunately, attempts to create a 3 dimensional cockpit instrument panel were unsuccessful. The canopy by nature is a three dimensional object, but the instrument panel has so many knobs, switches, and so much detail, that the Jack system simply cannot handle all the information. Instead, we have a flat 2 dimensional cockpit panel, with the instruments indicated by rectangles or squares, some of which are colored. Particular panels, such as the radio panel, can be selected for slightly more information, and the outlines of the knobs and selection switches can also be added for the price of speed and loading time. It is advisable to decide which section of the panel to focus on, and make that section of the panel more detailed, leaving the rest of the panel less detailed.

4.2.3 Environment Files

The following files in the /usr/local/upenn/demo directory are environment files:

- 95pilot.env This environment file has the 95th percentile male mannequin sitting in the full canopy cockpit with full color. This particular environment was used for colorization trials, and to see how far over the cockpit panel the pilot's head is at 95th percentile. This particular environment file is accessed by demo.jcl.

- ah64.env This environment file has the 50th percentile male mannequin sitting in the color cockpit with detailed radio panel. This particular environment isn't accessed by anything at all.

- radiosites.env This environment file has the 50th percentile mannequin sitting in the color cockpit with detailed radio panel, with sites defined around the radio panel and on the control surfaces of the collective and cyclic. Primarily used in the reach demo, this particular environment file isn't accessed by anything at all.

- radiosolid.env This environment file has the 50th percentile mannequin sitting in the color cockpit with detailed radio panel, radio and control sites defined, and with two lights illuminating the cockpit and the mannequin. For all intents and purposes, this file is identical to radiosites.env, except it allows us to painlessly shade the environment without having to creat and position lights beforehand. This particular environment file isn't accessed by anything at all.

- scene.env This environment file has the 50th percentile female mannequin leaning against the full canopy cockpit, checking out the interior. Originally intended as a reference point in space to be included in the occlusion-view demo, it created problems with the view from eyepoint and was left out for this phase. Instead, it served to prove a point that Jack can be used for any environment, including maintenance, and spawned an offshoot called outside.env that is typically brought up, shaded, and left on the console as a pretty picture. This particular environment file isn't accessed by anything at all.

- outside.env This environment file has the 50th percentile female mannequin leaning against the full canopy cockpit, with a Xth percentile female approaching from the other side of the aircraft's nose. Based on the scene.env file, the additional figure was added during a demo to display how easy the mannequins are to manipulate, and how quickly an environment can be created. Both this file, and scene.env, have a major fault in that they have a shading problem that solidifies a cockpit window where it's

not supposed to be shaded. This particular environment file isn't accessed by anything at all.

- sites.env This environment file has the 95th percentile male mannequin seated in the full canopy cockpit, with sites set on the radio panel, various places about the instrument panel, and on the control surfaces. Originally the test of the reach facilities, it has been superseded by radiosolid.env. This file is not accessed by anything at all.
- view.env This environment file has the 95th percentile male mannequin seated in the full canopy cockpit, with two cameras. One camera is the default camera, the one looked through when the environment is first brought up. The other camera is attached to the mannequin's left eye, allowing us to see what the mannequin would see when we attach to that camera and execute some movements. This file is primarily used by the view demo, and is accessed by view.jcl.

For information on how jcl and environment files access other files, please refer to the Jack User's Manual.

The "radiosites.env" and "radiosolid.env" files can be difficult to demonstrate reaches with, as the sites on the radio panel, cyclic and collective can be difficult to find. By going to the options menu, then the display menu, and selecting to turn segment sites on, all the sites that are defined on a figure can be displayed. To turn them back off, select the turn segment sites off from the display menu. Care must be taken when selecting a site to reach for; in particular, do *NOT* simply click once in the vicinity of where to reach to. If the wrong site is chosen accidentally, that is will be the site used. Instead, hold the mouse button down when clicking and carefully read the information presented in the blue message window. If the site is not the desired one the site can be changed by clicking once on another button on the mouse while still holding down the original button.

The file "scene.env" is not being accessed by any animation files at the moment because the animation files are having difficulty running from the correct viewpoint when the "scene.env" file is used instead of the "view.env" file.

The file "outside.env" is used primarily as an endpiece display for the 4D. Due to the problems with the shading, it is not yet possible to display it shaded, but it still looks good even in wireframe mode.

4.2.4 Animation Files

The following files in the /usr/local/upenn/demo directory are animation files;

- demo.jcl This animation file runs as follows: the camera is looking towards the cockpit and mannequin while the mannequin moves its head to the right, to the left, back to center, looks down at the panel, looks to the left side of panel, looks to the right side of panel, looks back to the middle of the panel, then looks up, looks down, looks up, looks down, looks to the left of the panel, and moves its left arm to touch the panel a few times.
- view.jcl This animation file runs exactly the same way demo.jcl runs, except that the view.env file has attached the camera to the mannequin's eyes. Thus, view.jcl

allows us to 'be' the mannequin and see what it sees, while demo.jcl allows us to watch the same movements from an exterior viewpoint.

For more information on how jcl files access other files, please refer to the Jack User's Manual.

The most basic commands in the animation files are movements of the mannequin's limbs. For more information on how to generate commands that the animation files will accept, refer to section 6.5 of this document. As bugs in the Jack program are fixed, the cockpit itself will be able to move with the pilot attached to the cockpit seat. In short, after some small problems with the animation system are fixed, full animation facilities will be available for producing complete animation sequences.

Currently all the animation scripts run in wireframe mode, as the facilities for running animation do not yet allow the animation to run in shaded mode.

5.0 NOTES

5.1 Limitations

Jack is sorely limited by the graphics and calculation power of the machine it runs on. Even though the CDE is capable of producing a three dimensional cockpit on the 4D, Jack spends far more than an acceptable period of time trying to load psurf files that contain substantial quantities of three dimensional information. For example, the CDE has several modes of display, ranging from simple squares delineating instruments to detail that includes the screws fastening instruments to the panel. Even in the middle range of display detail Jack has difficulty creating the cockpit. Instead, it is necessary to pick a particular object or panel that needs to be three dimensional and save it as a separate file to include with the rest of the less detailed cockpit.

The A3I project is also hampered by delivery times of the Jack product. Delivery of the tape is sometimes delayed, and additional problems necessitating re-compilation of the code can cause more time delay. The product itself, since it is still under development and enhancement, may be missing features, may still have bugs, or may have had a component changed that will not allow certain files to be run.

Additionally, there is a problem involving the lack of synchronization capabilities in Jack. While running the Symbolics and Jack together, it is necessary to synchronize the two. This necessitates the creation of some type of timing mechanism for Jack to be fully integrated with the tick-based A3I simulation.

5.2 Future Directions

Future versions of Jack will be required to solve a variety of problems. Whether these problems are solved by Upenn or by A3I staff is currently unknown. Regardless of who initiates the solutions, some features of Jack will be required in the near future:

- The ability to run animation in shaded mode. Currently animation only runs in wireframe mode, making it difficult to determine information about instrument visibility when the panel is viewed from the mannequin's eyepoint.
- The ability to communicate XYZ positions to the Symbolics or the Iris machines. Some XYZ information is needed by other machines running in the A3I demo, making it necessary to create a means of determining XYZ positions of the mannequin or environment and then be able to transmit this information to another machine.
- The ability to create a mannequin based on our own measurements of a human. Currently all of Jack's mannequins are built on pre-determined data and reflect percentile norms of a population. Should it become necessary to focus on the measurements of a particular human, the facilities to add and create this mannequin need to exist. This facility will also allow the project to select a subject, enter data into Jack and run simulations, and then take the subject to the environment and validate Jack's results.
- The ability to reflect force and accuracy in reach simulations. Once it has been determined that a position is reachable from the pilot's position, it becomes necessary to determine whether force or accuracy is more important when reaching a site. A greater or lesser degree of either may affect the outcome of the reach.
- The ability to detect imminent collisions and avoid them. When executing a reach, Jack moves along a straight line regardless of whether that line collides with another object or not. Some method must be developed to recognize objects in the reach path and detour around them rather than move through them.
- The ability to synchronize Jack with other simulations on a tick-by-tick basis. In order for the Symbolics to communicate with Jack in a timely manner, Jack and the Symbolics must be synchronized in some manner so that neither one is running faster than the other.

Future updates of Jack will be obtained either by tape or from downloading the relevant files over the Internet. The downloading option is being experimented with on a trial basis to determine its suitability for month-to-month updates.

6.0 USERS GUIDE

For a complete User's Guide to Jack, please refer to the Jack User's Guide referenced in section 2.1 of this document. Information on the foundations and framework of Jack can be found in the Programming with Jack manual referenced in section 2.1. Instructions on running the A3I demo files can be found in the JACK Demo Instructions and JACK Demo Speech, located in Appendixes C and D of this document.

6.1 Compiling Jack

Jack and its components have been written in C, and are therefore compiled using the Iris C compiler. Each directory containing source code for Jack comes with its own Makefile that compiles that directory and executes other commands necessary to build Jack. Jack has been written for a variety of different generation Iris machines, and variables particular to the host machine must be set before a compilation can be run. Appendix A gives an example of the instruction sheet accompanying Jack 3.5, including instructions on setting these variables and how to compile the system. See Appendix E for a detailed description of a compilation session with Jack.

Jack has been organized by Upenn into a particular sequence of directories. The "Programming with Jack" manual describes the various libraries, along with other information such as macros, variables, arguments, etc. The directory structure of Jack on the A3I 4D machine is displayed in Figure 2 as a directory breakdown. A more detailed listing of each subdirectory can be found in Appendix A. This particular directory structure has been chosen by UPenn and has been delivered in the manner displayed in Figure 2.

Level

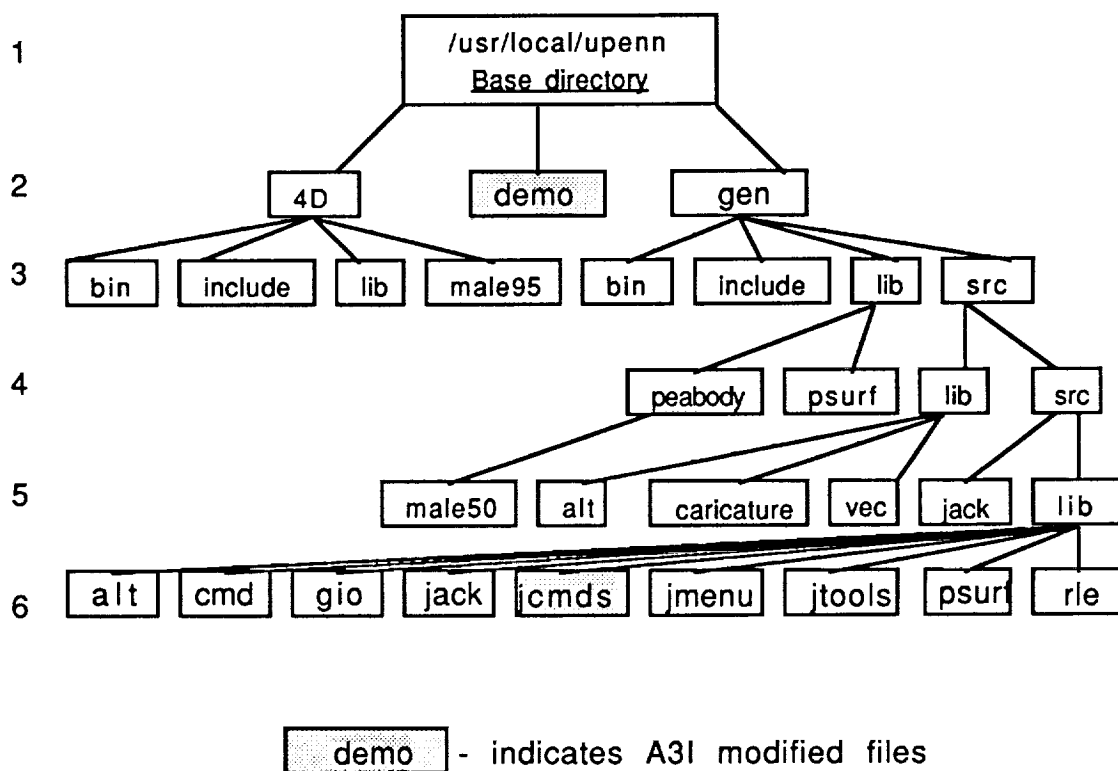


Figure 2: Jack Directory Breakdown

6.2 Porting Jack

It has been necessary in the past to port Jack to another Iris for demo or test purposes. To do this, the /usr/local/upenn directory must be copied to tape then reloaded onto the destination machine, and recompiled.

A few things to consider:

- Who do the Jack files belong to?
- Where are all the files?
- What type of machine is the software to be ported to? Does Jack support it?

Ownership: On the A3I 4DGT, all the Jack files are owned by user 'upenn' and group 'upenn', and the demo files are owned by user 'timelord' and group 'a3i_grph'. Since the 'timelord' and 'upenn' logins are not likely to exist on another machine, there are two options: 1) the sysadmin of the destination machine can create a 'timelord' and 'upenn' login with the same user and group id as 'timelord' and 'upenn' on the A3I machine (see below), or 2) the ownership of the files must be changed to something more generic that will correspond to an already existing login on the destination machine, such as 'guest'. This is best done before copying the files to tape.

```
guest:nu67SqHx9/X6:998:998:Guest Account:/usr/people/guest:/bin/csh
timelord:mJFHIObnUxYlg:24:30:G. Murdock Helms:/usr/people/u/timelord:/bin/csh
upenn:HHbXZveaq5D9w:50:50:UPenn Account:/usr/people/u/upenn:/bin/csh
```

Location: Figure 2 shows a detailed directory tree of where the Jack files reside. If /usr/local/upenn is copied to tape, the tape contains all the files necessary to run Jack.

Machine types: There are a variety of different operating system releases available for each Iris that may involve changes in the way the windowing system works. Check the status of the destination machine before attempting a port, and make sure the appropriate version of Jack is available for the destination machine. The version of Jack currently running on the A3I 4D will work on a 4D, a 4D60Turbo, the 4DGT, and the 4DGTx. Jack does not need to be recompiled when porting from the 4DGT to the 4DGTx.

See Appendix E for details on a sample port and compilation session from a 4D70T to a 4D70GT.

6.3 Modes of Operation

Jack has a variety of options that can be specified on the command line and will change the way Jack interacts with the user. A list of these options can be found in the Jack User's Guide, Chapter 1, Section 1.4.5, page 12. The most frequently used option is the -l option.

The -l option, also known as 'network mode', causes Jack to look at a specific file for its commands rather than watching the keyboard and mouse. Jack's code has been modified slightly so that if the -l option has been specified, a link with the Symbolics 3675 is initiated, and a communications package on the Symbolics must be set up for the two machines to communicate. Once the package has been set up properly, an environment can be read in and arranged to the user's satisfaction.

After this environment is displayed, Jack can be told to read from the command (script) file "symb.in" in the /usr/local/upenn/demo directory. Since the Symbolics is sending commands across the network to this file on the 4D, the Iris will execute each of the commands the Symbolics is sending it. If the link is being used to control environments involving reaches, it is necessary to predefine the sites to reach to before attempting the link. The most efficient way to do this is to create the environment with reach sites before initiating the link, and then loading that environment. For examples of the types of commands passed from the Symbolics to the Iris please refer to Section 6.5 of this document.

The only way to exit this mode is to kill the Jack program entirely, as Jack will not acknowledge any keyboard or mouse-entered commands once it starts reading the command file.

A detailed description of the changes made to UPenn's code and how it works can be found in Section 4.1 of the A3I Phase III Flight Dynamics / Guidance document listed in Section 2.

Jack currently does not have any special facilities to allow it to synchronize itself with events on the Symbolics. Instead, when Jack is ready to accept input from the Symbolics, it sends an "okay" message to the Symbolics, which issues a command to Jack and waits for Jack's next "okay" message. Jack accepts the command, executes it, and sends an "okay" message back to the Symbolics. There is a slight execution delay while Jack and the Symbolics send messages to each other, but it is not significant enough to cause problems with the demo.

6.4 Updating Jack

There are two ways to update the Jack software: by tape, or across the net. In the past, tapes have been mailed to us, which results in a week or more wait while the tape is made and shipped. A faster method which works well in emergencies is to download the files using FTP between the UPenn machine and the 4D.

The tape option, while simple, may take more time to arrive than is acceptable. Files may be missing, and there is a question about how to determine which old files may be removed without compromising the program.

The net option is immediately available, but is time consuming for the person transferring the files down. Cary Phillips, our point of contact, may be reached at (215) 898 1976 for the machine name, account name, and password needed to access and download the files.

Before beginning a download, the complete structure of the files on the UPenn machine must be understood so files are not accidentally misplaced. This usually requires an hour or two of looking around the UPenn machine while saving the session using the Unix "script" facility. Time is also a consideration here...files transferred in the middle of a West Coast day may not reach their destination intact, and it is best to wait until 9 or 10 pm PST before initiating a transfer session, as traffic will be way down at that hour. A complete familiarization with the Unix FTP facility is mandatory.

Most important to remember is that the machines should be fully backed up before the transfer, preferably within the previous week, and the existing Jack program and files should be fully backed up on tape before beginning the transfer. In case the update does not work, a copy of the previous working version of Jack can be retrieved from the tape.

Any alterations to UPenn source code must be copied and kept in a safe place to be compared to the new source code and to facilitate in updating the new code. This also includes new figure files, psurf files, or other files added to any part of the UPenn directory structure that has not been explicitly created by A3I. For example, the /usr/local/upenn/demo directory is an A3I directory, and does not map onto any other UPenn directory. Thus, during an update, no files in the demo directory will be affected or altered.

6.5 Creating Animation Scripts

In order to run either animation scripts, or to run Jack from the Symbolics, the instructions to Jack must be in a format Jack will understand. Since the menus are not utilized while reading from files, it becomes necessary to turn Jack's menu commands into text commands that can be put into a file. To do this, first decide what changes need to be made to the Jack environment. Next, go to the Options menu and choose "start JCL file". Jack will ask for a file name. Jack will then convert any of the subsequent menu commands to a textual format and save them in the file. The file will be closed when Jack is exited, and the file may be examined to determine the text version of each Jack command.

For those intending to work with the animation scripts at all, it is very important to learn how to handle the JCL script facility, which is covered briefly in section 3.12 of the Jack User's Manual. It is also helpful to understand how animation works. For each movement that a figure makes, there are many small movements (or "frames") that are made to accomplish that goal. As an example, the following sections describe how to move one of the pilots in the demo directory.

First, Jack needs a starting environment. Use the radiosites.env file, since it already exists. Once that environment is read in and displayed on the console, decide what to move in that environment. Since the pilot is what should be moving, the cockpit won't need to move. Now a decision must be made as to what will move on the pilot. Let's say we want to move the pilot's right shoulder so that his right hand will be sticking up in the air at roughly the same height as his shoulder.

First, take a look at the text in radiosites.env (Appendix B). We're looking for a figure, and we know that it's a mannequin. "pbmale50.fig" is the only figure in the file, and thus must be the pilot. By looking at the description of "pbmale50.fig", the figure's name is found to be "lee". Now look down the list of lee's joints until reaching the one that says right_shoulder. This line tells us what the displacement of lee's right shoulder is in X, Y and Z coordinates. It can be difficult to tell which axis is the one to change, so at this point it is advisable to experiment with the figure on the console. To adjust lee's shoulder, go to the main menu, choose the move menu, and select the adjust joint entry. Click left on lee's right shoulder, making sure to hold down the button to confirm that you have lee's right shoulder. By clicking on any mouse button, a red wheel will appear. Each mouse button corresponds to a different axis.

Depending on which axis has been chosen, the wheel will either be parallel to the horizon, or perpendicular, with the flat of the wheel running parallel to lee's chest, or perpendicular. To move lee's arm to the front and up from his body choose the wheel that is perpendicular to lee's chest and perpendicular to the horizon. By holding down the mouse button and moving the mouse, lee's arm can be swung up and down. Observe the yellow text and note which of the parameters changes as you move the mouse.

Since the rotating up and down of the right shoulder causes the numbers on the far right to change, these are the coordinates that will be affected. Some consideration must be given at this point as to

how far the arm should move. Let's pick 42 degrees as an arbitrary number. Now, decide how far to move the arm each "frame". Again, let's choose 5 as an arbitrary number, remembering that if too large a number is chosen, the intermediate movements will be jerky, but if too small a number is chosen, the intermediate movements will be smoother but will also take longer.

Now, go to the Options menu and start a jcl file called "testout". Then go to the Edit menu, choose Adjust Joint and pick the right shoulder. Move the right shoulder a short ways, and then quit out of Jack. Examining the file "testout.jcl" should reveal a line that appears similar to this:

```
adjust_joint("lee.right_shoulder",68.84deg,5.77deg,55.14deg);
```

The numbers in this example may not exactly match the numbers in your testout.jcl file. This is because your mouse movements may have been smaller or greater than the one used for this example.

Now start a different file. Using whichever UNIX text editor you're comfortable with, start a new file called "movarm.jcl". This will be the file that contains all the information Jack needs to read in an environment and perform animation. The first piece of information Jack needs to know is which environment to perform its animation in. Add to this file the line:

```
read_environment("/usr/local/upenn/demo/radiosites.env");
```

This causes Jack to load the radiosites.env environment. Now Jack must be told to start a binfile that will contain the 'compiled' version of the movements that it will be performing. Add the line:

```
init_binfile();
```

This tells Jack to start a binfile to save all the movements in. The name of the file will be given to Jack later. Now tell Jack what the first movement of the mannequin is. We already know the right shoulder's starting position from the environment file. From watching how Jack's arm moved earlier we know that the third parameter of the adjust_joint call is the parameter affected by moving Jack's arm forward and up. The first movement should be 5 more degrees than the starting position, so now add to the file the lines:

```
adjust_joint("lee.right_shoulder",68.84deg,5.77deg,53.14deg);  
add_frame_to_binfile();
```

We tell Jack to add the frame to the bin file since we want that movement to constitute the next frame of the animation. Continue adjust_joint and add_frame lines until the last adjust_joint command has incremented lee's right arm 42 degrees from it's starting position. After the corresponding "add_frame_to_binfile();" command, add the line:

```
write_binfile("movarm.bin");
```

This tells Jack to write the movements in order to the file named "movarm.bin".

The completed, short animation sequence now lives in the file named "movarm.jcl". To run the animation in Jack, exit and start Jack again, locate the "read jcl file" in the Jack command menus and type in "movarm.jcl" when Jack asks for the file to read. Jack will now read your animation file, perform the movements, and save an image of each frame in the file "movarm.bin". Once Jack has finished reading the animation file, exit Jack, start Jack up and execute the "load bin file"

command in the "playback" menu. By specifying the "movarm.bin" file, Jack will load the short animation frames into Jack and wait for the "playback" command to be executed.

If Jack does not move quite right, edit the animation file and add or subtract moves, convert it to bin format, and watch the animation run again. To produce a smooth, well-paced piece of animation you will spend lots of time breaking movements down into very small increments and continually re-running your animation script.

6.6 Generating Movements Automatically

Unfortunately, the chore of generating lots of lines of code to produce lots of little movements is quite tiresome. In an effort to help generate animation scripts quickly and with less effort, a C program called "Makescript.c" is currently being built to aid the scripting process. This program accepts information about which mannequin joint is to be moved, how far it is to be moved and which axis it will be moved along, and then produces a file that contains all the Jack instructions necessary to produce that movement. In its current working state, the program only alters one axis at a time and will not accept multiple joints to move. The program is being enhanced, debugged, and updated to handle movements along all three axes, as well as multiple joint movements. The code for the current working version of the program can be found in Appendix F.

6.7 Downloading from CDE

Once a CDE environment has been loaded into the CDE, it can then be downloaded to Jack. Simply start up CDE with the environment that will be downloaded to Jack. Then go to the CDE menu and choose Data Format. A window will appear at the bottom that asks for one of several types of file format. Select the Psurf format and click on OK. This window will go away and another window will appear in the center of the screen that will ask for a scaling factor. A 1 and a return should be typed in, unless the cockpit is to be scaled to a different size.

This window will be replaced by another window asking for a filename to be typed in. This is the filename that the psurf information will be saved in. This window will go away and another will appear in the upper left corner that has the legend "Select Bead Then Hit OK or Done" and three buttons...OK, Done, and Undo. Now, to choose a section of the environment that you want to save, simply click on that section so that a white line appears around the edge, then click on "OK". Each section will be saved in the order it is selected. When all the sections have been selected, clicking on "Done" will save all those sections in one psurf file.

The "Undo" button will erase the last input you clicked on. For more information on how to use the CDE facilities and bringing up environments, please refer to the A3I Phase III CDE Documentation listed in Section 2.1 of this manual.

6.8 Jack Installation Instructions

The following are instructions for installing Jack, and were included with the last Jack tape update.

Installing the Jack Software

When the software is extracted from the tape, it will create directories called "gen", "4D", and "3000". The "gen" directory

contains the 'general' source code and data files for Jack and its related programs. The "4D" directory contains files specific to the IRIS 4D, and the "3000" directory contains files specific to the IRIS 3000. In particular, there are "bin" subdirectories in each of these directories for executables for each of these machines. When you unload the tape onto a specific machine, one of the directories (4D or 3000) will be useless, depending upon what type of machine you have (or don't have). You should remove this directory to save space and confusion.

Several shell variables must be set in order to run and maintain Jack. The first is UPENN, which should be set to the "parent" directory where the tape was installed, e.g. "/usr/upenn". The other important variable is CPU, which should be either "4D" or "3000", depending upon the machine type. These should be set in "/etc/cshrc".

When installing this code on an IRIS 4D, you will also need to ensure that file \${UPENN}/\${CPU}/include/machine.h sets the proper preprocessor control for your machine. This file should contain exactly one of:

```
#define IRIS3000 1
or
#define IRIS4D60 1
or
#define IRIS4D70GT 1
```

If you are compiling the code for an IRIS 4D60, be sure that the proper symbol is defined!

After \$UPENN and \$CPU variables are set, "source" the shell script "\${UPENN}/gen/bin/upennenv.csh". This shell script sets the following shell variables:

```
PATH
INCLUDEDIR
CFLAGS
BINDIR
LIBDIR
UPENNMAKERULES
QDRAWLIB
PSURFLIB
PEALIB
IMAGELIB
HELPLIB
```

The PATH variable is set by "upennenv.csh" using the variables USRPATH and SYSPATH, which are the user part and system part of the path, respectively. These should also be set in "/etc/cshrc".

At Penn, our /etc/cshrc is as follows:

cut here

umask 022

stty erase "^H" kill "^U" intr "^C"

set term="iris-ansi"

setenv MAIL /usr/mail/\${LOGNAME}

setenv TZ EST5EDT

switch (\${LOGNAME})

case root:

setenv USRPATH ./etc

setenv SYSPATH /usr/local/bin:/usr/bin:/bin:/usr/bsd:/usr/sbin

setenv PATH \${USRPATH}:\${SYSPATH}

set prompt="# "

breaksw

default:

setenv USRPATH :\$HOME/bin

setenv SYSPATH /usr/local/bin:/usr/bin:/bin:/usr/bsd:/usr/sbin

setenv PATH \${USRPATH}:\${SYSPATH}

cat -s /etc/motd

if (`/bin/mail -e`) then

echo "you have mail"

endif

breaksw

endsw

setenv UPENN /usr/upenn

setenv CPU 4D

source \${UPENN}/gen/bin/upennenv.csh

cut here

To recompile Jack and its supporting programs, just "make" the directory "/usr/upenn/gen/src". This will in turn make the various libraries, putting them in "/usr/local/lib".

After you "make" the system, you may want to clean everything up by reissuing the "make" command with the argument "clean":

```
% cd /usr/upenn/gen/src/
```

```
% make clean
```

This will remove the unnecessary ".o" files. Unless you plan on doing programming using the Jack and Peabody libraries, you may also remove the object libraries from "/usr/local/lib"

You will find that the directory /usr/upenn/gen/lib/images contains some interesting images, but it also takes a lot of space. You may

delete these if necessary.

Any questions or problems, call Cary Phillips, 1-215-898-1976

7.0 APPENDICES

A. Jack Directory Detailed Breakdown

/usr/local/upenn:

4D	cshrc	gen	script.orig	typescript
README	demo	orig.reach	symb.out	
a.out	extra	ranlib	test.c	

Notes: This directory is the 'root' of Jack. All Jack files live under this directory.

/usr/local/upenn/4D:

bin	include	lib
-----	---------	-----

Notes: This directory is the 'information' directory of Jack for the 4D, hence the name. Each of the names under this directory is, itself, a separate directory.

/usr/local/upenn/4D/bin:

jack-3.5-4D

Notes: The 'jack-3.5-4D' file is the executable binary version of Jack that is created from compiling all the pieces of Jack.

/usr/local/upenn/4D/include:

machine.h

Notes: The 'machine.h' file is the information file that lets Jack know what type of machine it's dealing with.

/usr/local/upenn/4D/lib:

libalt.a	libgio.a	libjmenu.a	libpsurf.a
libcar.a	libjack.a	libjtools.a	librle.a
libcmd.a	libcmds.a	libpea.a	libvec.a

Notes: The files in this directory are created by compiling Jack, and are the archived library files Jack references.

/usr/local/upenn/demo:

95pilot.env	cockpit.pss	symb.in		Index	demo.bin
output.reach	view.bin		Makescript.c	demo.jcl	view.env
	radiosites.env		view.jcl		
ah64.env	figure.env	radiosolid.env	view2.jcl		
ah64.pss	reach4sites.jcl				
cockpit.env	male95	scene.env			
mov.bin	sites.env				

Notes: The files under this directory have been created at NASA for the A3I Project and are detailed in section 4.0 of this document.

```
/usr/local/upenn/gen:
bin          include      lib          src
```

Notes: 'gen' is the directory that is at the top of all the C source code for Jack. Each file under 'gen' is a directory itself, and each directory contains different information.

```
/usr/local/upenn/gen/bin:
cptree      init.sh~      ranlib          upennenv.csh~  vt100~
cshrc       lwpg          subdir          upennenv.sh
cshrc.usr   nextdir      tcd            upennenv.sh~
gmake       profile.csh  treecd         vt100
init.sh     profile.sh   upennenv.csh   vt100.csh
```

Notes: Information that Jack needs for environment and other data is located in this directory.

```
/usr/local/upenn/gen/include:
XtndRunsv.h grace.h      make.h         port.h         vec.h
attribute.h jack.h       pl            psurf.h
caricature.h jcmds.h     parsetab.h  rle_getraw.h
cmd.h       jmenu.h     peabody.h   svfb.h
gio.h       jtools.h    polhemus.h  svfb_global.h
```

Notes: This directory contains information Jack needs to compile the source code.

```
/usr/local/upenn/gen/lib:
peabody      psurf
```

Notes: This section of the 'gen' directory contains information for the graphics sections of Jack.

```
/usr/local/upenn/gen/lib/peabody:
body.fig      female99.fig  pbfemale25.fig  pbmale25.fig
female01.fig  male01.fig   pbfemale50.fig  pbmale50.fig
female05.fig  male05.fig   pbfemale75.fig  pbmale75.fig
female25.fig  male25.fig   pbfemale95.fig  pbmale95.fig
female50.fig  male50.fig   pbfemale99.fig  pbmale99.fig
female75.fig  pbfemale01.fig  pbmale01.fig
female95.fig  pbfemale05.fig  pbmale05.fig
```

Notes: This directory contains figure informatio for all the assorted genders and sizes of mannequins.

```
/usr/local/upenn/gen/lib/psurf:
axis.pss      cylinder.pss  medcube.pss  table.pss      unluparm.pss
bar.pss       floor.pss    nasapanel.pss tv.pss         unlupleg.pss
big.pss       ground.pss   plane.pss    unbhead.pss   unneck.pss
bigcube.pss  light.pss    poly.pss     unctorso.pss  unrclav.pss
camera.pss   link.pss     polybody.a   unit.pss      unrfoot.pss
```

ceiling.pss	link0.pss	puma.a	unitpoly.pss	unrhand.pss
chair.pss	link1.pss	pyramid.pss	unclav.pss	unrloarm.pss
chair1.pss	link2.pss	rect1.pss	unlfoot.pss	unrloleg.pss
chair2.pss	link3.pss	rect2.pss	unlhand.pss	unruparm.pss
cockpit.pss	link4.pss	skinny.a	unlloarm.pss	unrupleg.pss
cube.pss	link5.pss	sphere.pss	unlloleg.pss	
cyl.pss	male50	stand.pss	unltorso.pss	

Notes: This directory appears to be sample psurf files, test files, and information for Jack.

```
/usr/local/upenn/gen/src:
lib          src
```

Notes: Now we are getting to the serious source files. Each of these is a separate directory.

```
/usr/local/upenn/gen/src/lib
alt          caricature    vec
```

Notes: Again, each of these is a subdirectory.

```
/usr/local/upenn/gen/src/lib/alt:
Makefile     color.c         io.c          load.c
attribute.c  foo                  light.c       texture.c
```

Notes: This directory is mostly concerned with lighting sources.

```
/usr/local/upenn/gen/src/lib/caricature:
Makefile     depthlist.c    interface.c
antialias.c  intensity.c     lut.c
```

Notes: This directory has more lighting sources.

```
/usr/local/upenn/gen/src/lib/vec:
Makefile     device.c        intersect.c   postscript.c  tomatrix.c
chull.c      event.c         list.c       pseud.c       vector.c
color.c      feedback.c     matrix.c     quaternion.c
debug.c      findfile.c     msg.c        string.c
```

Notes: This directory has an assortment of files we have not yet connected to anything yet.

```
/usr/local/upenn/gen:
bin          include        lib          src
```

Notes: This directory contains more subdirectories.

```
/usr/local/upenn/gen/include:
XtndRunsv.h  grace.h        make.h       port.h        vec.h
attribute.h  jack.h         pl           psurf.h
caricature.h jcmds.h        parsetab.h  rle_getraw.h
cmd.h        jmenu.h        peabody.h   svfb.h
gio.h        jtools.h       polhemus.h  svfb_global.h
```

Notes: This directory contains information for many of the Jack commands.

```
/usr/local/upenn/gen/src
lib          src
```

Notes: This directory contains more subdirectories.

```
/usr/local/upenn/gen/src/lib
alt          caricature  vec
```

Notes: More subdirectories.

```
/usr/local/upenn/gen/src/src
jack         lib
```

Notes: And MORE subdirectories...

```
/usr/local/upenn/gen/src/src/jack
Makefile          a.out          main.orig
Makefile.orig  main.c          menu.c
```

Notes: This directory contains what appears to be the main program and executables for Jack.

```
/usr/local/upenn/gen/src/src/lib
Makefile  cmd      gio      jmenu      psurf
alt       errs     jack     jtools     rle
caricature files   jcnds     peabody    vec
```

Notes: Each of these are subdirectories with sources for their namesakes.

```
/usr/local/upenn/gen/src/src/lib/alt
Makefile  color.c  io.c      load.c
attribute.c  foo      light.c   texture.c
```

Notes: More sources for lighting.

```
/usr/local/upenn/gen/src/src/lib/caricature
Makefile  intensity.c  mknode.c  render.c
antialias.c  interface.c  mkpoint.c  scan.c
depthlist.c  lut.c        mkpoly.c   shade.c
geom.c       mkedge.c     print.c    update.c
```

Notes: More sources for lighting.

```
/usr/local/upenn/gen/src/src/lib/cmd
Makefile  device.c  helpfile  msg.c      screenmsg.c
cmd.c     help.c    menu.c    msg.old
```

Notes: We're not sure what's in here.

```
/usr/local/upenn/gen/src/src/lib/gio
Makefile      composite.c  gio.c          image.c
```

Notes: We're not sure what's in here either.

```
/usr/local/upenn/gen/src/src/lib/jack
Makefile      draw.c      look.c         moveview.c    trackwin.c
adjust.c      error.c      meter.c        peawin.c      transform.c
clip.c        event.c      meterwin.c     pick.c         view.c
colormap.c    genargs.c    mouse.c        picklist.c    window.c
describe.c    grid.c       move.c         project.c
docmds.c      highlight.c  movefigure.c   quit.c
dof.c         init.c       movenodes.c    script.c
domotion.c    jcl.c        movesite.c     snap.c
```

Notes: Here are a lot of windowing sources and other assorted things.

```
/usr/local/upenn/gen/src/src/lib/jcmds
Makefile      color.c      interplay.c    reach.c        shade.c
Makefile.orig create.c      jcl.c          read.c         system.c
adjust.c      delete.c     light.c        rename.c       view.c
attribute.c    display.c   movefigure.c   render.c       window.c
buffer.c       force.c*    moveitem.c     reroot.c       write.c
build.c        help.c      movesite.c     reset.c
collide.c      info.c      postscript.c   script.c
```

Notes: Here are a lot of sources for manipulating the mannequin.

```
/usr/local/upenn/gen/src/src/lib/jmenu
Makefile      color.c      force.c        object.c       shade.c
attribute.c    create.c     helpfile       options.c      bkg.c         csg.c         info.c
parameter.c    view.c
body.c         deformation.c light.c        playback.c     window.c
build.c        display.c    main.c         primitive.c    write.c
collision.c     edit.c       move.c         reach.c
```

Notes: Here are a lot of sources for assorted commands.

```
/usr/local/upenn/gen/src/src/lib/jtools
Makefile      buffer.c     interplay.c    postscript.c   shade.c
attribute.c    color.c      light.c        render.c
```

Notes: More sources for shading and rendering.

```
/usr/local/upenn/gen/src/src/lib/peabody
Makefile      find.c       mreach.c       parse.y        remove.c
attribute.c    interface.c  name.c         parseops.c     traverse.c
collide.c     keyword.c    new.c          print.c        update.c
create.c       lex.c        op.c           proplist.c     verify.c
defs.h         lex.l        parse.c        reach.c        write.c
expr.c         metric.c     parse.h        reachsite.c
```

Notes: Here are sources for a variety of commands, as well as other information.

```
/usr/local/upenn/gen/src/src/lib/psurf
Makefile      decompose.c  gen          patch.c      shade.c
bezier.c      draw.c          intersect.c  print.c      shadow.c
bin.c         edge.c          lex.l       reference.c  spec.c
clippoly.c    euler.c         new.c       render.c     util.c
curve.c       forward.c       normal.c    rp.y         verify.c
```

Notes: Here are more shading and drawing sources.

```
/usr/local/upenn/gen/src/src/lib/rle
Makefile      dither.c      rle_getrow.c rle_row_alc.c sv_putrow.c
Runsv.c       rle_getcom.c rle_putcom.c scanargs.c svfb_global.c
buildmap.c    rle_getraw.c rle_raw_alc.c sv_putraw.c
```

Notes: We've no idea what's in here.

```
/usr/local/upenn/gen/src/src/lib/vec
Makefile      device.c      intersect.c  postscript.c  tomatrix.c
chull.c       event.c       list.c       pseud.c       vector.c
color.c       feedback.c   matrix.c    quaternion.c
debug.c       findfile.c   msg.c       string.c
```

Notes: Here is a wide variety of assorted sources.

B. Sample Environment File: radiosites.env

```
attribute attribute5 {
    rgb = (1.00,1.00,0.00);
}
attribute attribute6 {
    rgb = (1.00,1.00,0.00);
}
attribute attribute7 {
    rgb = (1.00,1.00,0.00);
}
attribute attribute8 {
    rgb = (1.00,1.00,0.00);
}
attribute attribute9 {
    rgb = (1.00,1.00,0.00);
}
attribute attribute10 {
    rgb = (1.00,1.00,0.00);
}
attribute attribute11 {
    rgb = (1.00,1.00,0.00);
}
```

```
}  
attribute attribute12 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute13 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute14 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute15 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute16 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute17 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute18 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute19 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute20 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute21 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute22 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute23 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute24 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute25 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute26 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute27 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute28 {  
    rgb = (1.00,1.00,0.00);  
}
```

```
}  
attribute attribute29 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute30 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute31 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute32 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute33 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute34 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute35 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute36 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute37 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute38 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute39 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute40 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute41 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute42 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute43 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute44 {  
    rgb = (1.00,1.00,0.00);  
}  
attribute attribute45 {  
    rgb = (1.00,1.00,0.00);  
}
```

```

}
attribute attribute46 {
    rgb = (1.00,1.00,0.00);
}
attribute attribute47 {
    rgb = (1.00,1.00,0.00);
}
attribute attribute48 {
    rgb = (1.00,1.00,0.00);
}
attribute attribute49 {
    rgb = (1.00,1.00,0.00);
}
figure camera {
    attribute attribute2 {
        rgb = (1.00,1.00,0.00);
    }
    segment camera {
        psurf = "camera.pss";
        attribute = attribute2;
        site base->location = trans(0.00cm,0.00cm,0.00cm);
    }
}
figure usr_local_upenn_demo_cockpit {
    attribute dkgray {
        rgb = (0.42,0.43,0.42);
    }
    attribute red {
        rgb = (1.00,0.01,0.00);
    }
    attribute white {
        rgb = (1.00,1.00,1.00);
    }
    attribute green {
        rgb = (0.00,1.00,0.00);
    }
    attribute blue {
        rgb = (0.00,0.71,1.00);
    }
    attribute orange {
        rgb = (1.00,0.58,0.00);
    }
    attribute bluegray {
        rgb = (0.57,0.76,0.83);
    }
    segment usr_local_upenn_demo_cockpit {
        psurf = "/usr/local/upenn/demo/cockpit.pss";
        attribute = (dkgray,red,white,green,blue,orange,bluegray);
        site base->location = trans(47.01cm,30.16cm,-56.56cm);
        site rvol->location = trans(67.85cm,5.30cm,-78.58cm);
        site r1->location = trans(69.10cm,5.49cm,-73.81cm);
    }
}

```

```

    site r2->location = trans(69.48cm,3.64cm,-73.87cm);
    site rtrans->location = trans(68.70cm,-0.44cm,-78.10cm);
    site cyc->location = trans(47.20cm,1.61cm,-65.35cm);
    site clct->location = trans(29.39cm,39.94cm,-84.72cm);
}
}
figure ["pbmale50.fig"] ("polybody.a") lee;
figure lee {
    joint right_shoulder->displacement = (68.84deg,5.77deg,48.14deg);
    joint right_elbow->displacement = (44.49deg);
    joint right_wrist->displacement = (26.10deg,0.00deg,-6.46deg);
    joint left_shoulder->displacement = (19.56deg,25.61deg,22.25deg);
    joint left_elbow->displacement = (26.65deg);
    joint left_wrist->displacement = (26.10deg,-1.31deg,-1.59deg);
    joint right_hip_joint->displacement = (0.00deg,14.93deg,94.33deg);
    joint right_knee->displacement = (31.60deg);
    joint right_ankle->displacement = (0.00deg,0.00deg,-28.32deg);
    joint left_hip_joint->displacement = (0.00deg,8.19deg,90.85deg);
    joint left_knee->displacement = (26.55deg);
    joint left_ankle->displacement = (0.00deg,0.00deg,-26.85deg);
    joint right_clavicle_joint->displacement = (0.00deg,0.00deg);
    joint left_clavicle_joint->displacement = (0.00deg,0.00deg);
    joint waist->displacement = (0.00deg,0.00deg,0.00deg);
    joint atlanto_occipital->displacement = (0.00deg,0.00deg,0.00deg);
    joint solar_plexus->displacement = trans(0.00cm,0.00cm,0.00cm);
    joint right_knuckles->displacement = (62.44deg);
    joint left_knuckles->displacement = (90.00deg);
    joint right_ball_of_foot->displacement = (0.00deg);
    joint left_ball_of_foot->displacement = (0.00deg);
    joint base_of_neck->displacement = trans(0.00cm,0.00cm,0.00cm);
    joint right_sternoclavicular->displacement = trans(0.00cm,0.00cm,0.00cm);
    joint left_sternoclavicular->displacement = trans(0.00cm,0.00cm,0.00cm);
    joint right_eye->displacement = trans(0.00cm,0.00cm,0.00cm);
    joint left_eye->displacement = trans(0.00cm,0.00cm,0.00cm);
    joint root_torso->displacement = trans(0.00cm,0.00cm,0.00cm);
    joint root_rhip->displacement = trans(0.00cm,0.00cm,0.00cm);
    joint root_lhip->displacement = trans(0.00cm,0.00cm,0.00cm);
    segment left_toes->attribute = (attribute5,attribute6,attribute7);
    segment right_toes->attribute = (attribute5,attribute8,attribute9);
    segment right_fingers->attribute = (attribute5,attribute10,attribute11);
    segment left_fingers->attribute = (attribute5,attribute12,attribute13);
    segment left_foot->attribute = (attribute5,attribute14,attribute15);
    segment right_foot->attribute = (attribute5,attribute16,attribute17);
    segment right_lower_leg->attribute = (attribute5,attribute18,attribute19);
    segment left_lower_leg->attribute = (attribute5,attribute20,attribute21);
    segment right_upper_leg->attribute = (attribute5,attribute22,attribute23);
    segment left_upper_leg->attribute = (attribute5,attribute24,attribute25);
    segment lower_torso->attribute = (attribute5,attribute26,attribute27);
    segment center_torso->attribute = (attribute5,attribute28,attribute29);
    segment bottom_head->attribute = (attribute5,attribute30,attribute31);
    segment neck->attribute = (attribute5,attribute32);
}

```

```

segment right_clavicle->attribute = (attribute5,attribute33,attribute34);
segment left_clavicle->attribute = (attribute5,attribute35,attribute36);
segment right_upper_arm->attribute = (attribute5,attribute37,attribute38);
segment left_upper_arm->attribute = (attribute5,attribute39,attribute40);
segment right_lower_arm->attribute = (attribute5,attribute41,attribute42);
segment left_lower_arm->attribute = (attribute5,attribute43,attribute44);
segment right_hand->attribute = (attribute5,attribute45,attribute46);
segment left_hand->attribute = (attribute5,attribute47,attribute48);
segment right_eyeball->attribute = (attribute5,attribute49);
segment left_eyeball->attribute = (attribute5,attribute49);
}
constraint camera_root {
  connect world.base to camera.camera.base;
  displacement = xyz(-160.18deg,-71.09deg,-161.17deg) * trans(-177.49cm,87.41cm,-
64.85cm);
}
constraint usr_local_upenn_demo_cockpit_root {
  connect world.base to usr_local_upenn_demo_cockpit.usr_local_upenn_demo_cockpit.base;
  displacement = xyz(-89.91deg,0.00deg,0.00deg) * trans(41.76cm,75.78cm,-20.20cm);
}
constraint lee_base {
  connect world.base to lee.body_root.floor;
  displacement = xyz(-180.00deg,84.20deg,-180.00deg) * trans(-12.59cm,-54.90cm,14.94cm);
}

```

C. Jack Demo Instructions

Brief Overview:

First Demo:

- login
- start Jack
- create mannequins and demonstrate joint limitations
- exit Jack

Second Demo:

- start Jack
- bring up reach demo and store it

Occlusion:

- start Jack
- bring up animation demo
- demonstrate animation while attached to mannequin's eye
- point out occlusion of master arming switch by glare shield
- exit Jack

Reach Analysis:

- bring up stored reach demo
- demonstrate reach from shoulder (strapped to seat)
- replace mannequin's hand on collective
- demonstrate reach from waist (not strapped to seat)
- solidify environment

- leave solid environment onscreen for duration of demo

First Demo

1. Login on the 4D and run the correct .cshrc file.

I will set up an account for you on Coral that will contain the file 'jackshell'. After logging in, you must execute the command 'source jackshell'. There are paths and environment definitions that Jack needs that are set in the jackshell file, so don't forget to source it. If your prompt changes to 'tml@CORAL>', you're on your way.

2. Change directory to /usr/local/upenn/demo.

This directory contains all the files you will need to run the demo. Specifically, the files you want are 'radiosites.env' and 'view.bin'. To get to this directory easily, simply type in 'demo'.

3. Start up the Jack program.

Type in "jack" to start Jack.

4. Load the Mannequins.

(this is the first section in the demo. I have doubts whether it will remain, but I'll put it in anyways.)

- a. when the blue window at the bottom says, "Press mouse button for main menu", move the mouse over the "Create Menu" selection until it turns black, move it to the right until the "Create Menu" appears. Now choose the "Bodies" selection, and move right until you get a list of bodies. Run the the mouse button down and release on the "95th percentile polybody male".

- b. Jack will want you to name the male. His default name is "Fred", so you can just hit return.

- c. When the figure comes up, you need to move him over to make room for the female. Click right on "Edit Menu", move the mouse to get the "Move Menu" from the Edit Menu, and release the mouse button on "Move Figure".

- d. Choose the figure to move by clicking on some portion of it.

- e. Move the figure by clicking on the left mouse button and shifting the mouse left until the male is sufficiently off to one side that the female can be loaded.

- f. Hit escape to exit the "move figure" mode.

- g. Load the female figure by following step one, only this time stop and release on "95th percentile polybody female".

- h. You will need to change the name this time. The "delete" key serves as a backspace key, so erase "Fred" with the delete key and type something else in, then hit return.

- i. To demonstrate joint limitations, go to "Edit Menu", "Move Menu", and release on "Adjust Joint". Choose any joint on either figure...the elbow joint works well, though it only has one direction to move, and the shoulder joint shows all three degrees of freedom.

- j. Once you've chosen the joint you want, press a button on the mouse. If a red wheel appears, move the mouse and watch the limb. It will not move past a pre-set point that defines the limit of human movement.

- k. To exit the "adjust joint" mode (You can only adjust *one* joint at a time!), press escape.

- l. To adjust another joint, follow step i, and pick another joint to play with.

5. Exit Jack

To exit the program, which you need to do to clear the screen, click on "Quit", and again on "Quit".

Second Demo:

1. Loading the Reach Demo:

a. when the blue window at the bottom says, "Press mouse button for main menu", press the right mouse button and hold it down. When the menu appears, slide the arrow over the 'options menu' selection so that it turns black, and then slide the mouse to the right so the options menu will appear. Now slide the mouse down to the 'background menu' option, and slide it right again so the background menu will appear. Now slide the mouse down so that 'bkg off' turns black, and release the mouse button. The background grid should now disappear.

b. Click right again for the main menu, slide to 'create menu' and then slide right to select 'read environment'. Release the button. The window at the bottom will now ask you for a file. Type in 'radiosolid.env'. Should you make a mistake, the 'delete' key will back up and erase. Press return to start loading the file. It will take several minutes to load this file, so you want to start loading as soon as you can.

c. Store this window. To store this demo, click left in the close box of first the jack window and then blue message window. They will sleep peacefully until you bring it back again.

2. Loading the Animation Demo:

a. Move the arrow back over your original login window. Change directory to 'male95' (type in 'cd male95'). Please note that you MUST be in the 'male95' directory for the animation demo to load correctly. Once you are in the 'male95' directory, start up Jack again by typing in 'jack'.

b. Shut off the grid by selecting 'options', 'background', and releasing on 'background off'.

c. Now load the animation demo by selecting 'utilities', 'playback', and 'load bin file'. The blue window will now ask for a file. Type in './view.bin'. This demo will also take several minutes to load.

Occlusion:

1. Running the Animation Demo:

a. Recite speech about attaching view to mannequin's eyepoint.

b. Click right and select "utilities", "playback" and "playback with view". Jack will now execute animation with the viewpoint attached to Jack's eyes.

c. Point out tailwheel lock and main armament switch are obscured by glare shield and are commonly mistaken for each other.

d. You are done with the animation. Click right, and select "quit", then choose "Yes, quit" to exit Jack.

Reach Analysis:

1. Running the Reach Demo

a. click left in the middle of the small box that the blue window made when you closed it.

b. click left in the middle of the small box that the main Jack window made when you closed it.

c. The cockpit will come up with all the colors skewed, so you first need to reset the colors. Select "edit", "attributes" and "change color", and click on any line of the cockpit or the mannequin.

d. A small red box will appear. Jack wants you to create a new window for the color meters. Click right, drag the mouse to make a box, and click right again. The window will materialize, and Jack will *automatically* reset all the colors for you.

e. Make sure the arrow is inside the new color window, and press the escape key. The background of the color window should now turn black.

f. Move the arrow up to the gray bar that says "color meters". It should turn into a circle. Click right, and release on "Push".

g. Now move the arrow up to the gray bar that says "Jack window". It again should turn into a circle. Click right, and release on "Pop".

h. Now you're ready to try a reach. First, select "edit" from the main window, then "reach menu", then "interactive reach".

i. Jack wants you to pick a point to touch. You want to find the site on the radio transmitter....roughly the center of the transmitter button. If you don't find it at first, keep trying.

j. Once you've found the site, it will display red XYZ lines. Now Jack wants you to type in a value, so simply hit return to use the default.

k. The next step is to select Jack's left fingers, which can be tricky. Try to press and hold on the junction of all the lines at the tip of Jack's fingers. Don't let go of the button until you're sure you clicked on the right place. You will see red XYZ lines, and yellow lines describing what you've clicked on. If the lines say "lee.left_fingers" near the end, you can let go of the button. If they say something else ("clct" may appear...that's the collective), click another button to select the next closest site.

l. Now Jack wants the last joint allowed to move during the reach demo. You want to select his left shoulder in the same manner you selected Jack's hand, by holding the button to make sure you have the right site. Once you've selected his left shoulder, Jack will execute the reach. Jack will also tell you how far away you are.

m. Now, put Jack's hand back on the collective. The collective site can be tricky to find...you need to practice finding it. It's about 1/3 of the way up on the slotted handgrip of the collective, on the outside (far left). Hit return to enter the default value, and then click on Jack's left hand, then on his shoulder, to make him reach for the collective again.

n. Now....make Jack reach for the radio transmitter again, only moving from the waist! Follow steps i through l, making sure to choose Jack's waist instead of his shoulder this time.

o. Solidify Jack. Click on "Utilities Menu" in the main menu, slide right to "Shade Menu", then release on "Shade", and Jack will become solid.

D. "Makescript.c", Movement Generating Program

Introduction

The original concept for Makescript stemmed from the enormous quantities of time it took to produce a working animation file that had any sense of smoothness to it. Animation files in Jack consist entirely of Jack commands in text format that are read into the program and then executed. After the sequence of commands is executed, they are saved to a 'bin' file that can be read in and executed much faster than the execution that happens during the reading of the file.

The primary problem with writing these files is that a simple movement of the mannequin, such as raising and then lowering an arm, can not be specified in such a way that only requires the user to state the start position and the end position of the arm. Instead, the movements had to be broken down into a smaller sequence of movements that would appear smooth when played back. No facilities were available in Jack to input the joint, the start position and the stop position, and which

axis would be affected. Instead, Jack users were left with the option of editing hundreds of lines of code themselves to produce small increments of motion.

From the sheer amount of time spent in editors creating movements Makescript was born as a utility for Jack users. The program is extremely simplistic, clumsily coded, and somewhat redundant in the current version. Future versions of Makescript will be more compact, understandable and elegant.

Sections

Makescript has three basic sections which are also its three functions: GetInput, Calculate, and WriteFile.

Functions and Arguments

GetInput

GetInput is the first function. It takes five arguments from the user and passes them on to other function(s) which it calls later.

The first argument 'joint', which is a character string, is the name of the joint to be moved. The user does not specify the mannequin's name, as the program assumes it is the mannequin named 'lee' and automatically inserts that name into the resulting animation script. The program does not spellcheck the input to make sure the joint exists. There is checking done to determine whether the user's joint has one degree of freedom, three degrees of freedom, or if the joint is base_of_neck which needs transformation information. Depending on what type of joint has been input, GetInput will ask for different information from the user.

The second argument 'temp', which is also a character string, is which axis the movement will take place on. This input is then checked to make sure the user has entered X, Y or Z rather than a non-existent axis that will cause the animation to fail. If the user has entered an incorrect axis, the program will not go further until X, Y or Z is entered.

The third argument, 'first', is an integer and contains the starting position of the joint.

The fourth argument, 'last', is an integer and contains the ending position of the joint.

The fifth argument, 'amount', is an integer and contains the value by which the arm should move each frame.

Also included are integer variables that contain values for non-changing axes ("axis1" and "axis2"; values for transformations that do not change ("trans1", "trans2", "trans3"); and a flag to determine when correct axes are input ("test"); and two character strings, one of which reads in the X,Y or Z input for checking before transferring it to coord ("temp"); the other string is set to the axis that movement is performed around("calcaxis").

GetInput asks for joint, axis, start, end and amount information, then calls Calculate and passes along the values of calcaxis, axis1, axis2, bodypart, first, last, amount, trans1, trans2 and trans3.

Calculate

Calculate accepts ten values from GetInput, uses them to determine if movement is positive or negative around the axis, then begins calculating joint increments and calls WriteFile after each calculation to store the information.

The first argument, 'rotate', contains the axis that movement will occur around.

The second argument, 'nomove1', contains the integer value of a nonmoving axis.

The third argument, 'nomove2', contains the integer value of the other nonmoving axis.

The fourth argument, 'part', contains the name of the mannequin's joint that will be moved.

The fifth argument, 'start', contains the integer value of the beginning position of the joint.

The sixth argument, 'stop', contains the integer value of the finishing position of the joint.

The seventh argument, 'degree', contains the integer value that determines how much the movement will be incremented each time.

The eighth argument, 'tx', contains the integer value of the unmoving X transformation.

The ninth argument, 'ty', contains the integer value of the unmoving Y transformation.

The tenth argument, 'tz', contains the integer value of the unmoving Z transformation.

In addition, Calculate has two integer variables, one of which serves as an alterable counter for the calculate loops ("counter"); the other records counter's value each time through the loop ("temp").

Calculate checks whether the increment is positive or negative, then enters a loop so that the first value is the value the user has entered, the next value is that start value incremented by the amount the user wanted, and so on until the last calculation is that of the end position. For each time through the loop, as each increment is calculated, Calculate calls WriteFile to store these numbers and passes to it rotate, nomove1, nomove2, part, counter, tx, ty and tz.

WriteFile

WriteFile takes eight arguments that are passed to it by Calculate and uses them to correctly format each command line and prints that line out to a default file.

The first argument, 'move', is a character designating which axis movement will take place around.

The second argument, 'norot1', is the value of the first of the two non-rotating axis.

The third argument, 'norot2', is the value of the second non-rotating axis.

The fourth argument, 'piece', is the joint of the mannequin that will be moved.

The fifth argument, 'alter', is the altered value of the movement.

The sixth argument, 'xtrans', is the unmoving X transformation.

The seventh argument, 'ytrans', is the unmoving Y transformation.

The eighth argument, 'ztrans', is the unmoving Z transformation.

WriteFile also has special pointers to allow it to open a file and write movement information to it.

WriteFile first opens the default output file "jack.comm" to write information to. Then it checks to see what type of joint is being moved. If the joint is base_of_neck, it requires extra transformation information. For each type of joint, the axis being moved is checked so that the changing value is placed in the correct spot each time. Each time WriteFile is finished executing, it closes the input file.

Problems

Makescript does not support multiple axis movements. Nor does it provide for the complete creation of a new script, and is designed instead to build segments of animation code that must be added to an already existing file by hand. Once a joint's movements have been computed and written, Makescript quits and does not allow for the input of another joint. The program will not take more than one joint at a time to move.

Code

```
#include <stdio.h>
/*  GM Helms, Oct. 12, 1988
    Makescript.c -> a program to produce JACK jcl files
    given joint information and increment information.

    Variables:
    axis        which X, Y or Z axis to change
    joint        which joint to alter
    alpha        starting position angle of joint
    omega        ending position angle of joint
    increment    how far to increment each movement

    Functions:
    GetInput      takes initial setup values
    Calculate      calculates each change in joint angle
    WriteFile      writes each change to an output file
*/

/* GetInput -->    test is a flag for the x,X,y,Y,z,Z testing loop. If
                   test is 0, one of the correct axis were entered. If
                   test is 1, the entry was not an axis and needs to be
                   entered again.
*/
GetInput(coord, bodypart, first, last, amount)
char coord;
char bodypart[50];
int first, last, amount;
```

```

{
int test, axis1, axis2, trans1, trans2, trans3;
char temp[2]; /*patch for reading /n in buffer if not xy or z first time */
char calcaxis; /* patch to let print routine know where moving axis is */

test=1;
printf("                JACK Script Generating Program\n");
printf("\nWhich joint on the mannequin will be moved? ");
scanf("%s", bodypart);
if (bodypart[5]!='k' && bodypart[6]!='k' && (bodypart[5]!='e' && bodypart[6]!='l')
&& (bodypart[6]!='e' && bodypart[7]!='l') )
{
printf("\nWhich axis, (x, y, or z) do the changes take place on? ");
while (test != 0)
{
scanf("%s", temp);
switch(temp[0])
{
case 'x':
case 'X':
test=0;
calcaxis='x';
break;

case 'y':
case 'Y':
test=0;
calcaxis='y';
break;

case 'z':
case 'Z':
test=0;
calcaxis='z';
break;

default:
printf("\nEntry must be x, y or z axis. Please re-enter.");
test=1;
break;
}
} /* end while loop */
} /* end of if not a single-joint bodypart*/
else
calcaxis='x';
coord=temp[0];
printf("\nWhat is the start coordinate? ");
scanf("%d", &first);
printf("\nWhat is the end coordinate? ");
scanf("%d", &last);

```

```

printf("\nBy what amount should the coordinates change per move? ");
scanf("%d", &amount);
if ( (bodypart[5]!='e' && bodypart[6]!='l') && (bodypart[6]!='e' && bodypart[7]!='l')
&& bodypart[5]!='k' && bodypart[6]!='k' )
{
    switch(temp[0])
    {
        case 'x':
        case 'X':
            printf("\nEnter unchanging Y coordinates: ");
            scanf("%d", &axis1);
            printf("\nEnter unchanging Z coordinates: ");
            scanf("%d", &axis2);
            break;
        case 'y':
        case 'Y':
            printf("\nEnter unchanging X coordinates: ");
            scanf("%d", &axis1);
            printf("\nEnter unchanging Z coordinates: ");
            scanf("%d", &axis2);
            break;
        case 'z':
        case 'Z':
            printf("\nEnter unchanging X coordinates: ");
            scanf("%d", &axis1);
            printf("\nEnter unchanging Y coordinates: ");
            scanf("%d", &axis2);
            break;
    } /* end of case */
} /*end of if */
else
{
    axis1=0;
    axis2=0;
}
if (bodypart[0]=='b')
{
    printf("\nEnter unchanging X transformation: ");
    scanf("%d", &trans1);
    printf("\nEnter unchanging Y transformation: ");
    scanf("%d", &trans2);
    printf("\nEnter unchanging Z transformation: ");
    scanf("%d", &trans3);
}
else
{
    trans1=0;
    trans2=0;
    trans3=0;
}

```

```

        Calculate(calcaxis, axis1, axis2, bodypart, first, last, amount, trans1, trans2, trans3);
    }

/* Calculate    takes axis variable and figures each increment, then calls
                WriteFile and passes all the info and the incremented moves
                to be printed out.
*/

Calculate(rotate, nomove1, nomove2, part, start, stop, degree, tx, ty, tz)
char rotate;
int nomove1, nomove2, start, stop, degree, tx, ty, tz;
char part[50];
{
    int counter,temp;
    counter=0;

    if (degree>0)
        for (counter=start, temp=counter-degree; counter<=stop; counter=temp+degree)
        {
            temp=counter;
            WriteFile(rotate, nomove1, nomove2, part, counter, tx, ty, tz);
        }

    if (degree<0)
        for (counter=start, temp=counter-degree; counter>=stop; counter=temp+degree)
        {
            temp=counter;
            WriteFile(rotate, nomove1, nomove2, part, counter, tx, ty, tz);
        }
}

/* WriteFile    prints information out to a file in acceptable Jack format
*/

WriteFile(move, norot1, norot2, piece, alter, xtrans, ytrans, ztrans)
char move;
int norot1, norot2, alter, xtrans, ytrans, ztrans;
char piece[50];
{
    FILE *input_file, *fopen ();

    input_file = fopen ("jack.comm", "a");
    if (input_file == NULL)
        printf ("\n--> jack.comm could not be opened.<--\n");
        if (piece[0]=='b')
        {
            if (move=='x')
            {
                fprintf(input_file, "\nadjust_joint(\"lee.%s\",xyz(%ddeg,%ddeg,%ddeg) *
trans(%dcm,%dcm,%dcm));", piece, alter, norot1, norot2, xtrans, ytrans, ztrans);
            }
        }
    }

```

```

        fprintf(input_file, "\nadd_frame_to_binfile();\n");
    }
    if (move=='y')
    {
        fprintf(input_file, "\nadjust_joint(\"lee.%s\",xyz(%ddeg,%ddeg,%ddeg) *
trans(%dcm,%dcm,%dcm));", piece, norot1, alter, norot2, xtrans, ytrans, ztrans);
        fprintf(input_file, "\nadd_frame_to_binfile();\n");
    }
    if (move=='z')
    {
        fprintf(input_file, "\nadjust_joint(\"lee.%s\",xyz(%ddeg,%ddeg,%ddeg) *
trans(%dcm,%dcm,%dcm));", piece, norot1, norot2, alter, xtrans, ytrans, ztrans);
        fprintf(input_file, "\nadd_frame_to_binfile();\n");
    }
}
else if ( (piece[5]=='e' && piece[6]!='y') || (piece[6]=='e' && piece[7]!='y') || piece[5]=='k' ||
piece[6]=='k' )
{
    fprintf(input_file, "\nadjust_joint(\"lee.%s\",%ddeg);", piece, alter );
    fprintf(input_file, "\nadd_frame_to_binfile();\n");
}
else
{
    if (move=='x')
    {
        fprintf(input_file, "\nadjust_joint(\"lee.%s\",%ddeg,%ddeg,%ddeg);", piece, alter,
norot1, norot2);
        fprintf(input_file, "\nadd_frame_to_binfile();\n");
    }
    if (move=='y')
    {
        fprintf(input_file, "\nadjust_joint(\"lee.%s\",%ddeg,%ddeg,%ddeg);", piece, norot1,
alter, norot2);
        fprintf(input_file, "\nadd_frame_to_binfile();\n");
    }
    if (move=='z')
    {
        fprintf(input_file, "\nadjust_joint(\"lee.%s\",%ddeg,%ddeg,%ddeg);", piece, norot1,
norot2, alter);
        fprintf(input_file, "\nadd_frame_to_binfile();\n");
    }
}
fclose (input_file);
}

main()
{
    char axis;
    char joint[50];
    int alpha, omega, increment;

```

```
}      GetInput(axis, joint, alpha, omega, increment);
```

Annex E

Army-NASA Aircrew/Aircraft Integration Program

A³I

**Software Detailed Design Document:
Training Assessment**

prepared by

Carolyn Banda

December 1988

Table of Contents

1.0	INTRODUCTION.....	E-1
1.1	Identification.....	E-1
1.2	Scope.....	E-1
1.3	Purpose.....	E-1
2.0	RELATED DOCUMENTATION.....	E-2
2.1	Applicable Documents.....	E-2
2.2	Information Documents.....	E-2
3.0	REQUIREMENTS AND DESIGN APPROACH.....	E-3
3.1	Requirements, Methods, and Rationale.....	E-3
3.2	Hardware Environment.....	E-4
3.3	Software Environment.....	E-4
4.0	DETAILED DESIGN DESCRIPTION.....	E-6
4.1	Organization.....	E-6
4.2	Unit Detailed Design.....	E-8
4.2.1	Task Data Collection Program (TDCP).....	E-8
4.2.1.1	TAS-input function.....	E-9
4.2.1.2	Get-obj function.....	E-9
4.2.1.3	Get-obj-characteristics function.....	E-9
4.2.1.4	Attribute value input functions and menus.....	E-9
4.2.2	Training Assessment Module (TAM).....	E-10
4.2.2.1	TAM Control Rules.....	E-11
4.2.2.2	Input Functions.....	E-12
4.2.2.3	Learning Experience Rules.....	E-12
4.2.2.4	Media Rules.....	E-13
4.2.2.5	Training Time Rules.....	E-13
4.2.2.6	Output Rules.....	E-14
4.2.2.7	Output Functions.....	E-15
5.0	NOTES.....	E-15
5.1	Miscellaneous.....	E-15
5.2	Limitations and Future Directions.....	E-15
6.0	USERS GUIDE.....	E-17
6.1	Overview.....	E-17
6.2	How to run the Task Data Collection Program (TDCP).....	E-17
6.2.1	Loading TDCP.....	E-17
6.2.2	Running TDCP.....	E-18
6.3	How to run the Training Assessment Module (TAM).....	E-18
6.3.1	Booting the ART world.....	E-18
6.3.2	Getting around in ART.....	E-19
6.3.3	Loading TAM.....	E-19
6.3.4	Running TAM.....	E-19
6.3.5	Justification with ART.....	E-20
7.0	APPENDICES.....	EA-1
A.	Glossary of Terms, Acronyms, or Abbreviations.....	EA-1
B.	Module Hierarchy Charts.....	EB-1
B.1	Task Data Collection Program (TDCP).....	EB-2
B.2	Training Assessment Module (TAM).....	EB-3
C.	Sample Task Data File.....	EC-1
D.	Sample Displays.....	ED-1

Table of Contents

D.1	Task Attribute Input Displays from TDCP.....	ED-1
D.2	Sample task object with characteristics.....	ED-4
D.3	Sample rules.....	ED-5
D.4	Sample agenda.....	ED-7
D.5	Output screen.....	ED-8
D.6	Justification networks.....	ED-9

1.0 INTRODUCTION

1.1 Identification

This document establishes the requirements and detailed design of the Training Assessment Computer Software Configuration Item (CSCI), which forms a part of the A3I Computer Program System. Descriptions of the detailed processing requirements, structure, I/O, and control are provided for each lower level Computer Software Component (CSC), unit, or function contained within the CSCI.

1.2 Scope

The material in this document is directed toward three categories of readers: 1) those who wish to learn what Training Assessment in A3I does, 2) those who wish to use the Training Assessment software to predict training requirements for a set of tasks, and 3) those who might want to modify and update the Training Assessment software. Familiarity with ART (Automated Reasoning Tool by Inference Corp.), Common Lisp (especially object-oriented programming), and the Symbolics environment is assumed.

1.3 Purpose

The purposes of the Training Assessment software are twofold: 1) to give the equipment designer early feedback about the training consequences of his/her design and mission and 2) to provide assistance to the training designer by aiding the development of a quasi-POI (Program Of Instruction). To do this, the Training Assessment software predicts training requirements for a set of mission tasks in the form of learning experiences, media, and training times necessary to train an incoming student with a specified experience/training level to successfully perform each task. At this point, we simply assume "successful" performance; we do not predict different training requirements for varying levels of performance. The standard of performance is assumed to be embedded within the task/objective (terms we use synonymously) and particularly within the task characterization data. Training cost is not yet computed but could be easily added if cost per hour were known for the various training media.

The training requirements assigned are sensitive both to incoming student training level, which is expressed as degree of familiarity with the tasks and their associated equipment, and to budget level, which can be set to low, medium, or high. Training times are adjusted based on the media selected, so that, for example, a complex task can be trained more quickly with a dedicated part task trainer than with a less costly, but less appropriate alternative training medium.

A comparative analysis can be done by examining the training requirements for two or more sets of equipment designs and mission tasks.

An explanation facility is available through ART's justification capability to show diagrammatically how the resulting "quasi-POI" was derived. For any selected fact, the justification network, which is a directed graph, shows all the rules which fired to assert this fact and all the facts which caused these rules to fire.

2.0 RELATED DOCUMENTATION

2.1 Applicable Documents

Air Force Regulation 50-8, Training Policy and Guidance for ISD, 1983. (Media descriptions)

ART Programming Tutorials, Volume 1: Elementary ART Programming and Volume 3: Advanced Topics in ART, ART Version 3.0, Inference Corp., Los Angeles, California, 1987.

ART Reference Manual, ART Version 3.0, Inference Corp., Los Angeles, California, 1987.

Flight Training Guide for AH-64 Aviator Qualification Course, United States Army Aviation Center, Fort Rucker, Alabama, February 1988.

Smith, Barry and Banda, Carolyn, "Use of Knowledge-Based System to Assess Aircrew Training Requirements as Part of Conceptual Design", paper in progress, 1989.

Training Analysis Support Computer System (TASCS) User's Guide, prepared by Logicon, Inc., San Diego, CA for Air Force Systems Command, ASD/YWB, Wright Patterson AFB, Ohio, August 1987.

2.2 Information Documents

Branson, Robert K., Rayner, G.T., Cox, J.L., Furman, J.P., King, F.J., and Hannum, W.H., Interservice Procedures for Instructional Systems Development (6 vols.) (TRADOC Pam 350-30 and NAVEDTRA 106A). Ft. Monroe, VA: U.S. Army Training and Doctrine Command, August 1975.

Ellis, John A. and Wulfeck II, Wallace H., Handbook for Testing in Navy Schools, Navy Personnel Research and Development Center, San Diego, California, October 1982. (An explanation of IQI (Instructional Quality Inventory) is given in Chapter 2: Classification of objectives and implications for testing.)

Symbolics Manual 7A: Programming the User Interface - Concepts, Cambridge, Mass., 1988.

3.0 REQUIREMENTS AND DESIGN APPROACH

3.1 Requirements, Methods, and Rationale

As we stated in Sec. 1.3, the primary goal of the Training Assessment software is to provide both equipment designers and training designers early feedback as to the training implications and requirements during the conceptual design stage so that design decisions and/or mission features which have a negative impact on training requirements can be examined and modified in software rather than in hardware. The impact of planned student changes can be examined as well. To achieve this goal, it was decided to partially automate the process of using task analysis and ISD (Instructional Systems Development) methods to predict training requirements.

First, we describe historical background for the training assessment area. The Phase III training assessment software represents a refinement of the Phase II work done in this area. The approach of developing a quasi-POI (Program Of Instruction) was felt to provide the best way of defining and measuring training requirements for a set of tasks to be trained. This approach was first suggested by Dr. Charles Jorgensen during Phase II planning. ISD was chosen because 1) it is in current use by the military to assess training needs, and 2) it is well documented and provides a step-by-step method for developing a POI from job requirements. The Phase II effort, which was our first attempt to apply ISD, followed the approach outlined in the ISD manuals by Robert Branson et al. Each task, or learning objective, was placed in a learning category, of which there are twelve. Four learning categories are primarily psychomotor in nature, seven are primarily cognitive, and one is attitudinal. We did not use the attitudinal category in Phase II. With each learning category is associated a two-dimensional table with task characteristics and training considerations on one axis and training systems on the other. Based on each task's characteristics (including cueing, feedback, and response requirements), a table match was done to find any and all of the training systems which could be used to train the task. A general budget level was indicated for each training system so that the user could select the less costly training system if more than one was applicable.

For Phase III the progression was made from a table-based method to a rule-based approach for determining training requirements. The Phase III Training Assessment Module (TAM) capitalizes on work done by Logicon in developing a database-oriented program called TASCs (Training Analysis Support Computer System). TASCs uses ISD methods to perform task analysis and assign training requirements based on task characteristics and some training design expertise which is encoded in the program in the form of IF...THEN rules. A good deal of TASCs's training assessment must be done manually by the user. In order to automate this process for A3I's MIDAS (Man-machine Interface Design/Analysis System) workstation, we decided to make certain simplifying assumptions. We assume tasks are equivalent to learning objectives, at least for the set of tasks we are analyzing. Currently with TAM, there is no inherent organization in the task set; future plans call for organizing the task structure into a hierarchical tree of learning objectives. In addition, TAM constructs a partial Program Of Instruction (POI) instead of the more complete POI provided by TASCs. TASCs builds lesson elements in the form of sets of learning experiences, media, and time to train for specified learning objectives, then carries the training design process to the point of grouping these lesson elements into lessons and organizing lessons into modules. Considerable user intervention is required for this process. TAM, however, stops at the point of assigning lesson elements to each learning objective. These lesson elements form the beginning of

a quasi-POI; the word "quasi" is used because TAM's output does not presently form a complete POI composed of sequenced lesson modules.

TAM's set of rules represents an expanded capability in other respects, however, in that TAM takes into account student background and training budget level in its analysis. Rules to assign learning experiences were taken from TASCs, with minor modification; part task learning experience assignment depends on student experience with the equipment. Media rules were obtained from Barry Smith, who has expertise in the area of training design. The media rules incorporate the following philosophy for making media assignment sensitive to budget level:

- 1) The lowest fidelity media are assigned, regardless of budget, which satisfy all particular cueing requirements and characteristics for a given task,
- 2) Even if the budget doesn't permit it, high fidelity (and costly) media are assigned for certain highly difficult, safety or mission critical tasks, and
- 3) If the budget doesn't permit the appropriate media assignment for required cueing or other characteristics, and the task does not fall into one of the highly critical areas referenced in 2), the actual weapon system is assigned as the training media.

The basic time-to-train formulas were taken from TASCs with some minor modification; these formulas use various task attributes, such as difficulty level, time to perform, safety criticality level, mission criticality level and, in the case of cognitive part task training, a multiplication factor based on the demands imposed by the task's learning subcategories and reasons for difficulty. An "ideal" time to train is computed using these formulas, one for each applicable learning experience type; included in this computation is a multiplication factor which takes into account the student's degree of familiarity with the task and equipment. Next, an "actual" training time is computed by adjusting "ideal" time based on the media assigned.

Ten varied aircrew tasks from the actual AH-64 training program were used as test cases to validate TAM's output. These tasks ranged from purely flying oriented tasks to aircraft system emergency procedures. Over the 10 tasks, 28 of the 29 learning experiences predicted by TAM for the tasks examined are present in the current AH-64 training program. Media predictions matched those currently in use for 23 of the 29 learning experiences. The exceptions were all in the academic portion of each task. The training time predictions were more difficult to validate, in part because the AH-64 training program did not list separate times for the individual tasks, but rather gave all the tasks trained during each training day. Comparing times for tasks trained on day number 9 and day number 32 of the Apache program showed TAM's results to be 50% less and 10% less than actual training time, respectively. However, the Apache data most likely includes a certain amount of overhead. For more details on the validation runs, please see "Use of a Knowledge-Based System to Assess Aircrew Training Requirements as Part of Conceptual Design" by Barry Smith and Carolyn Banda.

3.2 Hardware Environment

The Training Assessment software runs on the Symbolics 3600 series under Genera 7.2. To run ART, a minimum of 8 megabytes of memory is recommended.

3.3 Software Environment

There are two separate training assessment programs for Phase III -- the Task Data Collection Program (TDCP) and the Training Assessment Module (TAM), which are described in detail below. These two programs currently run in a stand-alone mode; the user first collects task data offline with the TDCP, then runs TAM to predict training requirements for selected tasks.

The task data collection program is written in Symbolics Common Lisp (Genera 7.2); TAM is written in ART (Automated Reasoning Tool) and Symbolics Common Lisp. ART is a rule-based expert system building tool written by Inference Corp.

TAM can be run in interpreted form (slow to load; rule text available for browsing) or in compiled form (fast to load; rule text not available for browsing).

ART provides capability to explain results and conclusions through its justification facility.

4.0 DETAILED DESIGN DESCRIPTION

4.1 Organization

The following diagram (Fig. 1) shows data and control flow for the two training assessment programs: (1) task data collection program and (2) TAM (Training Assessment Module).

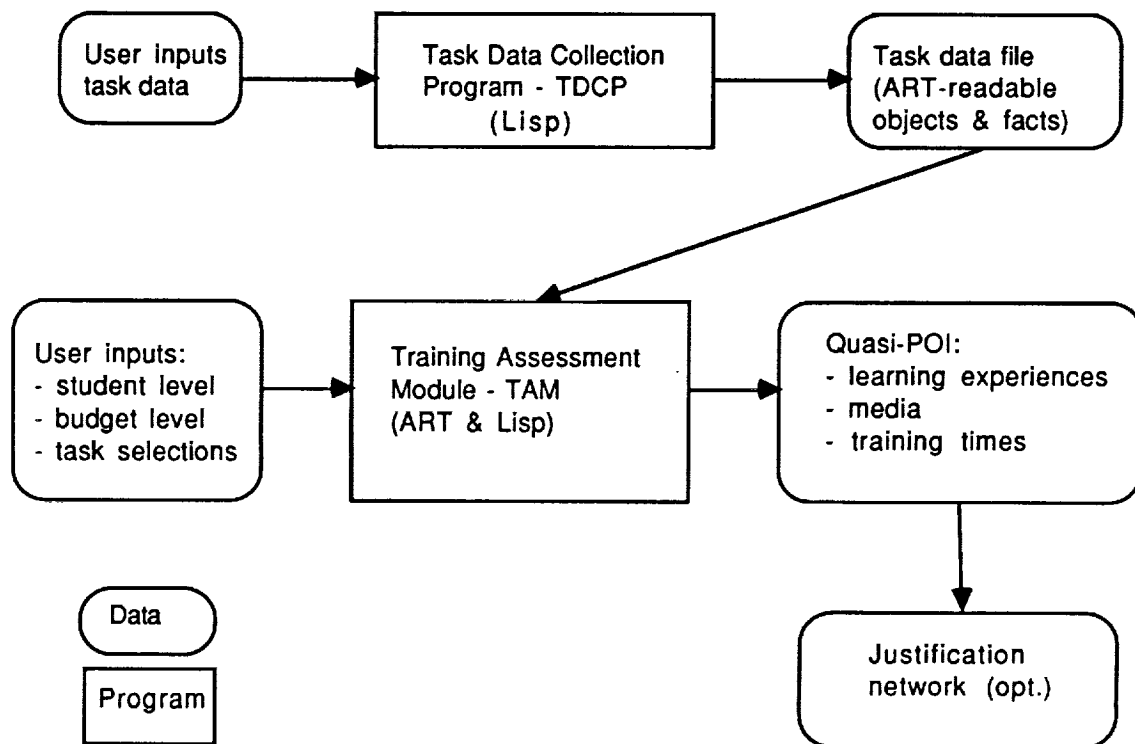


Fig. 1 TRAINING ASSESSMENT: CONTROL AND DATA FLOW

The following brief explanation of how to operate the training assessment software describes the two programs (task data collection program and TAM) and how they relate to each other.

To obtain training requirements for a set of tasks, the user runs the menu-oriented task data collection program to enter a set of mission tasks and their associated characteristics, such as cues, systems used, reasons for difficulty, a subjective difficulty rating, mission and safety criticality ratings, frequency of task performance, and learning subcategories as well as degree of familiarity with the tasks and equipment for students from a variety of backgrounds. The task data collection program writes this information to a file, organizing the tasks (or rather learning objectives) into a set of ART objects and facts in a format readable by TAM. Next the user runs TAM from the ART

environment to predict training requirements for a set of tasks, a specified student training level, and a selected budget level. Various rules in TAM are activated according to task characteristics to determine for each learning objective, in succession, a set of required learning experiences, media, and training times. Total training time for each task is computed, as well.

Module hierarchy charts showing functional organization within each program appear in Appendix B; these diagrams show function calling sequences and, in case of TAM, rule progressions. The file location of each function appears below its name in parentheses.

A description of the location of the source code files for the two programs follows; note that both programs are loaded via main load files.

1) Task data collection program files are in directory b:>carolyn>tas>ui. Relevant files are:

TDCP-loadfile.lisp	- load file for task data collection program
input-defs.lisp	- menu definitions for user interface
input-driver.lisp	- control functions

The following files contain menu definitions and functions to query the user for specified task characteristics.

types.lisp	- inactive; reserved for future use. This attribute could be used to categorize tasks along some dimension, such as learning objectives vs enabling objectives.
systems.lisp	- the aircraft systems which are involved in performance of this task
cues.lisp	- detailed selection of cues required in task performance
rfd.lisp	- reasons for difficulty
scrit.lisp	- reasons for safety criticality; safety criticality level is automatically assigned based on these reasons
mcrit.lisp	- mission criticality rating
frequency-time.lisp	- frequency of task performance and time to perform
lscs.lisp	- learning subcategories for this task; includes cognitive, psychomotor, and misc. categories.
newness.lisp	- degree of familiarity with this task and its equipment for the set of students defined in the variable *student-list* in the file input-driver.lisp.
TAM-task-set.art	- This file is constructed by the task data collection program and contains task data in the form of ART-readable objects and facts.

For a description of current values which these task attributes can assume, please see Appendix D, which shows sample screens from the Task Data Collection Program.

2) TAM (Training Assessment Module) files are in directory: b:>carolyn>TAM; relevant files are:

tam-loadfile.lisp	- load file for TAM; this is loaded from the ART environment.
setup-fns.lisp	- supporting Lisp functions to obtain from user input parameters for TAM

output-fns.lisp	- supporting Lisp functions to assist print TAM results to screen
TAM.art	- definitions for relations, schemata, facts, and driver rules
le-rules.art	- rules to assign learning experiences
media-rules.art	- rules to assign media
time.art	- rules to compute ideal training times; also rules to adjust ideal times to actual training times
output-with-ART.art	- rules to print results to screen
TAM-task-set.art	- input file of data objects; written by the task data collection program.

4.2 Unit Detailed Design

4.2.1 Task Data Collection Program (TDCP)

The TDCP is a stand-alone, menu-oriented program written in Symbolics Common Lisp to take advantage of predefined menu types which provide extensive menu capability. Basic menu types include pop-up menus, multiple choice menus, and choose variable values facilities; these menu types are combined through flavor combination to produce more complex menus. Available menu types are explained in Symbolics Manual 7A: Programming the User Interface - Concepts.

The TDCP allows the user to enter and characterize a series of tasks by selecting all applicable values for a set of attributes for each task. TDCP employs a "mixed initiative" style; it is up to the user to ensure that appropriate values for all attributes are entered for each task. It should be noted that the Task Data Collection Program was developed for internal use only. It is functionally complete; however, its user interface is not totally polished and also is not totally "user robust" although a careful user should have no trouble using it.

The TDCP was designed to minimize user entry error by having the user select displayed values for the task attributes rather than type them in; the only "typing" required is entry of the task name. In addition to task attribute values, the user is asked to indicate for each task its degree of familiarity for each student in the planned student set. The possibilities are: familiar with both the task and equipment, with the task only, or with neither. A list of planned student types resides in the top level file (input-driver.lisp) in the variable **student-list** and in the variable **student-item-list** in the TAM file setup-fns.lisp. The current student set contains UHT graduates and Apache pilots; this can be modified by changing **student-list** in input-driver.lisp and **student-item-list** in the TAM file setup-fns.lisp.

The TDCP is designed in a modular way to allow the user to add new attributes or attribute values in the future. To add a new attribute, one would create a new file and construct a menu to display its possible values; one would also modify existing rules in TAM which reference the new attribute and its values. Knowledge of Lisp and Zmacs would be required; however, templates exist for a variety of menu capabilities in the current set of TDCP files.

Function calling sequences, which are also indicated in the module hierarchy chart in Appendix B, are as follows:

```
TAS-input
  get-obj
```

- get-obj-characteristics
 - get-cues
 - get-types
 - get-systems
 - get-rfd
 - get-difficulty-rating
 - get-rfsc
 - get-mission-crit-rating
 - get-frequency
 - get-time-to-do
 - convert-to-minutes
 - get-lscs
 - get-safety-crit-rating
- get-newness-values
- reformat-newness-list-entry

Descriptions follow for the main functions of the TDCP; the remaining functions, most of which obtain values for a task attribute, have straightforward logic. Screens showing menu displays produced by these functions appear in Appendix D. For more information on these functions, the reader is referred to the code listings in Appendix E.

4.2.1.1 TAS-input function

The *TAS-input* function is the main driver function for the TDCP; it contains a loop to obtain for each task desired, the task name, its attribute values, and its degree of familiarity for the planned set of students. To terminate the task entry process, the user enters "nil" in response to the query for task name. After all tasks and values have been entered, *TAS-input* reformats the data into ART-readable object and fact definitions and writes these definitions to a task data file.

4.2.1.2 Get-obj function

This function is very simple; it uses the system function *tv:choose-variable-values* to ask the user to enter the task name, which it returns to its caller, *TAS-input*.

4.2.1.3 Get-obj-characteristics function

Get-obj-characteristics, called by *TAS-input*, is responsible for obtaining from the user a set of values for each task attribute for a given task. To do this, it has a case statement which allows the user to enter task attribute values in any order. When the user has entered all applicable values for all attributes for the current task, he or she clicks on "DONE" to proceed to the next task.

4.2.1.4 Attribute value input functions and menus

This set of functions uses menus to query the user for task attribute values. Current attributes include: types (not used), systems, cues, reasons for difficulty (rfd) and difficulty rating, safety criticality (scrit), mission criticality (mcrit), frequency, time to perform, and learning subcategories (lscs).

These functions and menus will be described as a class since they use the same basic methods with minor variations. Variable and function names are in italics. Menu types used by TDCP functions include:

- pop-up-multiple - used when multiple values are allowed and expected, specifically in gathering values for the attributes types (not in current use), systems, cues, reasons for difficulty, reasons for safety criticality, mission criticality rating, and learning subcategories.
- pop-up - used when only one value is allowed, in obtaining a value for frequency and "newness" (degree of familiarity for a given student).
- tv:choose-variable-values - actually a function, but it creates a menu for selecting one value from a set; default value is in boldface. Used to obtain a value for difficulty rating and time-to-do.

The following description applies to pop-up-multiple menus and associated functions; however, use of pop-up menus are very similar. The designator *<attribute>* means attribute name, for example, cues, systems, etc. First **<attribute>-menu** is defined as well as **<attribute>-list**, which contains a list of possible attribute values. A variable called **<attribute>-choices** is defined to designate options after all applicable attribute values have been selected and to name the functions which will be called when the user clicks on the various options. Typically the options are "Do it", meaning "use the selected values", or "None", indicating no values are selected. The corresponding functions are *<attribute>-do-it*, which sets the variable **selected-<attribute>** to a list containing all values highlighted by user, and *<attribute>-none*, which sets **selected-<attribute>** to nil. The menu itself, called **<attribute>-menu**, constructed using a "setq" to define menu characteristics such as menu type, default character style, label, border width, etc.

The function which drives the whole process is called *get-<attribute>*; it sets up the label to contain current task name, displays the menu, deactivates it, and returns values chosen by the user in the variable **selected-<attribute>**.

4.2.2 Training Assessment Module (TAM)

Most of TAM is written in ART, with Symbolics Common Lisp used for its menu capabilities to allow the user to select student background, budget level, and tasks for the current run. Several Lisp functions also assist with formatting the output.

TAM's knowledge about instructional design is embedded in its rules for assigning learning experiences, media, and training times. In addition, program control is accomplished through a set of "driver" rules and output control is performed through a set of output rules, which collect results and print them in order according to learning experience. Currently TAM contains about 100 rules, of which 77 are concerned with instructional design and the rest with control, input, and output. The breakdown within TAM is: 21 rules to assign learning experiences, 31 to assign media, and 25 to compute training time (8 for ideal training time and 17 for actual training time).

TAM's module hierarchy chart appears in Appendix B. TAM's rules are grouped according to function and TAM's Lisp functions appear in calling sequence. Again, file locations for functions and rules appear below their names in parentheses.

Training assessment for a given task is a sequential process. First, a set of learning experiences (or instructional strategies) is assigned based on task characteristics. The possible learning experiences are: three types of explanation (textual, graphic, and dynamic), demonstration, three types of part-task training (cognitive, psychomotor, and affective), and full task training. At a minimum, one type of explanation (and only one) is assigned, as is full task training. Optionally, based on task characteristics and student background, any or all of the remaining learning experiences may be assigned. Learning experience assignments are based on task characteristics and student background. Media assignments are based on task characteristics, learning experience category, and budget level. Training times are based on task characteristics, learning experience category, training medium, and student background.

After a task has its set of learning experiences, a training medium is chosen for each learning experience. Available training media include: textbook/workbook, lecture, videotape, videodisc-CBT (Computer Based Training), cockpit familiarization trainer, cockpit procedures trainer, operational flight trainer, part-task trainer, weapon system trainers with and without motion platforms, as well as actual system. These media types span the major classes of training devices presently in use.

Finally, a training time is computed for each lesson element in a two step process. First, an "ideal" training time is computed, with student background taken into account; this "ideal" time is then adjusted to an "actual" time, based on medium used.

Salience values for certain rules are used during inferencing when it is important to control the order of firing. For example, this is done in the assignment of media within the various categories of learning experiences where a "default" method of reasoning was necessary to emulate the domain expert's line of reasoning. For example, in the case of demonstration and full task training learning experiences, salience values allow TAM to assign media with successive levels of fidelity, starting at the low end (cockpit familiarization trainer) and finally assigning the actual system as a default if no other medium could be assigned.

4.2.2.1 TAM Control Rules

Several rules, which reside in the file TAM.art, handle the startup and control of TAM. COLLECT-TASKS gathers all tasks which were read in when the task data file was loaded and places them in a list (*entire-task-list*). START-UP is activated when the user enters "(TA)" to start the training assessment process; START-UP calls functions to obtain input information required by TAM--namely, student background, budget level, and task set for which training assessment is to be done. START-UP asserts the chosen task set as a list and causes INITIATE-TRAINING-ASSESSMENT to activate. INITIATE-TRAINING-ASSESSMENT forms the main control loop over the task list by asserting the "find-training" relation for each task in the list, one at a time.

The rule CASE1-OLD-TASK-OLD-EQUIPMENT retracts "find-training" for any task if the student is familiar with both that task and its equipment.

4.2.2.2 Input Functions

Several functions, which reside in the TAM file *setup-fns.lisp*, display menus to obtain from the user input parameters required by TAM: *get-student-background*, *get-budget-level*, and *select-tasks*. These functions display selection menus so the user need not type in values by hand; this reduces chances for errors. The list of selected tasks returned from the menu in *select-tasks* requires further manipulation to reduce it to a simple list of tasks; this is done by the function *reduce-user-select-list*.

4.2.2.3 Learning Experience Rules

The rules to assign learning experiences were taken directly from TASCs; additionally, however, TAM takes into account student familiarity with task and equipment by assigning part task training only if both task and equipment are new to the student.

As for programming considerations, saliences are used with explanation rules to assign explanations in order with dynamic explanation first, graphic second, and textual last. This is done because dynamic subsumes the other two, graphic subsumes textual, and textual is assigned if the other two do not apply. There is also a tradeoff between training assessment runtime vs. explainability of results. In a number of learning experience cases, most notably demonstrations, there are multiple rules for assigning the learning experience. During training assessment, all applicable rules are placed on the agenda. In the case of the demonstration learning experience, for example, 5 or 6 rules might apply. These rules could check to see if demonstration has already been assigned and remove themselves from the agenda in that case, but the information that they also were activated would be lost. It was decided to let them go ahead and fire (even though they won't assign demonstration again if it already has been assigned) so that they will appear in the justification network. The fact that multiple rules were activated could lend strength to the conclusion that a demonstration was needed.

4.2.2.4 Media Rules

The media rules, as mentioned before, were developed by Barry Smith; he also devised a media-learning experience assignment matrix, which appears below in Figure 2. Learning experiences are from TASCs and media types and their implied functionality are from AFR 50-11, Management of Aircrew Training Devices.

LEARNING EXPERIENCES

MEDIA	Text Exp	Graph Exp	Dyn Exp	Demo	Cog PTT	Psy PTT	Att PTT	FTT
Textbook/Workbook	X	X						
Interactive Slide/Tape	X	X						
Lecture w/Visual Aids	X	X						
Video Tape	X	X	X					
Video Disc/CBT	X	X	X		X			
CFT				X				X
CPT				X	X			X
OFT				X	X	X		X
Dedicated PTT					X	X		
WST w/o Motion				X	X	X		X
WST with Motion				X	X	X		X
Actual System				X	X	X	X	X

Figure 2

The rationale for each rule is given in a comment above the rule in the code listings in Appendix E.

4.2.2.5 Training Time Rules

A two-step process is used to compute training time. First, an ideal training time is computed using formulas from TASCs with some adjustments. The original formulas follow; the TASCs manual contains rationale for them. Note that all times are in minutes.

explanations: difficulty * 60
 demonstration: difficulty * time-to-do * 1.5
 cognitive ptt: difficulty * time-to-do * safety-criticality * lsc-factor * .2
 psychomotor ptt: difficulty * time-to-do * safety-criticality
 affective ptt: safety-criticality * 10

full task training: $\text{time-to-do} * (\text{difficulty} * \text{difficulty} + 1)$

The "lsc-factor" is a multiplier based on estimate of cognitive difficulty of the task; both learning subcategories and reasons for difficulty are taken into account in the rules which determine its value. These rules assign a value from 1 to 4 on the following basis: if the task involves using unaided concepts, procedures, rules, or principles and reasons for difficulty include a) decision on mission or situation, b) coordination with other ship or ground party, or c) few or vague cues, a value of 4, 3, or 2 is assigned, respectively; otherwise, lsc-factor is set to 1.

Some of the existing multiplication factors have been modified and some new ones added, notably a factor to account for student familiarity with task and equipment. These factors are subject to change as further testing is done; for current values, please see code listings for the file time.art in Appendix E. One multiplication factor is noteworthy; currently set to a value of one, it is present in all the formulas to compute ideal training time. It can be used to reduce or increase all training times roughly proportionally.

After ideal training time has been computed, the second step occurs: the actual training time is computed by adjusting ideal time with a multiplication factor whose value depends on the medium assigned. For example, if budgetary restrictions require use of the actual system instead of a dedicated part-task trainer for a psychomotor task, a multiplication factor increases training time since the medium is than optimal.

4.2.2.6 Output Rules

TAM output is somewhat involved; nine rules control the scheduling and printing of each task and six rules, one for each learning experience type, print actual results within each task. Notification that a partial result (i.e., one learning experience) for a task is ready for printing is done by ASSERT-DUPLICATE-RESULT, which asserts a "print-training" fact complete with all relevant results after actual time has been computed. The duplication is necessary because we want to save the original "training-time-actual" fact but we need to delete the extra (equivalent) fact during the printing process. The rule PRINT-SUMMARY has minimum salience so it will not fire until all results for all tasks have been posted; when it does fire, it asserts the "ta-setup-output" fact, which triggers SETUP-OUTPUT-WINDOW. SETUP-OUTPUT-WINDOW reshapes windows, sets up fonts and triggers PRINT-TITLE-AND-HEADER by asserting the "ta-headers" fact. After printing title and headers, this rule initiates printing of task results by asserting the "print-TA-results" fact. This starts up PRINT-RESULTS-OUTER-LOOP, which iterates through the task list, identifying current task, one at a time. The training time sum is also set to zero here. COLLECT-TASK-RESULTS is activated for the current task and schedules all results for the current task by asserting "print-training*" facts for them; this rule also sums training time as it collects these results. COLLECT-TASK-RESULTS retracts the "print-training" facts for the current task as it asserts the new "print-training*" facts so that TAM can tell when it has collected all facts for the current task by checking for the absence of "print-training" facts for that task. This condition is detected by SCHEDULE-CURRENT-PRINT-TASK, which asserts a "print-marked-results" fact for the current task. The "print-marked-results" facts cause the applicable print rules to activate, with their order controlled by saliences so that results will be printed in order according to learning experience type: explanation, demonstration, cognitive part task training, psychomotor part task training, affective part task training, and finally full task training. A rule with lower salience, PRINT-TOTAL-TIME, prints the total training time for the task.

4.2.2.7 Output Functions

Several small, straightforward functions are used to assist with display of TAM's results. *Compute-leading-spaces* allows TAM to right-justify training times for tabular printout, and *display* improves output appearance by removing hyphens from object names. For example, *send-radio-message* becomes "SEND RADIO MESSAGE".

5.0 NOTES

5.1 Miscellaneous

Several lessons were learned related to setting up an ART program so that the justification network gives a complete answer to the question: "Why was this fact asserted into the knowledge base?" One must keep intermediate facts in the knowledge base to provide a trace of how results were obtained; you can't always "clean up" by eliminating intermediate results. Also, when assignments are made by default (that is, by checking for absence of facts in the knowledge base) as we do when assigning media, the justification network does not show that we checked for the absence of those facts.

Another programming consideration concerns checking for absence of some attribute value for attributes with multiple values, such as cues, reasons for difficulty, etc. One must use (not (cues dynamic-visual)) in the rule conditions to check for absence of dynamic visual cues rather than (cues ~dynamic-visual), because the latter matches every "cues" fact for the task which is not dynamic-visual. This condition arises only for multi-valued attributes.

5.2 Limitations and Future Directions

The process of task collection is not automated; the user must enter manually each task with its associated characteristics. It is possible that this process could be automated, at least in part, by acquiring tasks directly from the mission activity history list and examining their performance and context to deduce such characteristics as the nature of their cueing, frequency, time to perform, and learning subcategories. Other characteristics, critical for the analysis, such as reasons for difficulty, may be more difficult to infer. This area requires further investigation. Also, it may be difficult to construct the learning hierarchy automatically from the mission decomposition tree since the two are not identical. Automatic generation of enabling objectives would require a great deal of embedded training design expertise.

However, if task data collection continues to be done manually, it should be noted that the task data collection program is modular and it would be fairly simple to add new task attributes or new values for a given attribute should the need arise in the future; some Lisp knowledge would be required.

A future direction which would yield valuable information for the training assessment module as well as the MIDAS workstation as a whole is the development of a metric for operational device complexity. This metric could be used, in part, to assess level of difficulty for a task (which is required by TAM).

Currently, TAM does not handle a hierarchy of learning objectives (tasks); adding this feature would greatly enhance TAM's results.

The ISD process could be carried further beyond simply assigning lesson elements for each task to be trained; lesson elements could be grouped into lessons, which could then be grouped into modules.

A rough cost of training could be calculated by adding to TAM the approximate cost per hour of the media in the media pool.

A student skills model could be developed so that student skill level is not directly linked to each task description, as is currently done. Student skill level would be compared to required skill level and the analysis would predict training required to bring student from current skill level to required skill level.

Continuation training and skill retention are not assessed currently.

In the future, if the training assessment is to be enlarged to handle a comprehensive set of mission tasks, a true database should be used for task data so that a large number of tasks can be easily handled, accessed, and modified by the user. Currently tasks reside in sequential order in a source file created by the task data collection program.

TAM provides a framework into which domain-dependent rules can be inserted. A user could insert rules embodying his or her own philosophy for assigning learning experiences, media, and training times. Cost computations could be easily added to the rule-based system. At present, a knowledge of the Symbolics editor Zmacs and ART syntax are required to add or modify rules.

6.0 USERS GUIDE

6.1 Overview

Predicting training requirements in Phase III is a two-step process. First, the user sets up a file containing a set of tasks with their associated characteristics by loading and running the Task Data Collection Program; the user then loads the Training Assessment Module (TAM) and the task data file and runs TAM, which prints training requirements to the screen. Many different runs can be made from the same task data file since the user selects input parameters and tasks for each run.

To show flow of control and data between the two programs, Figure 1 is repeated below.

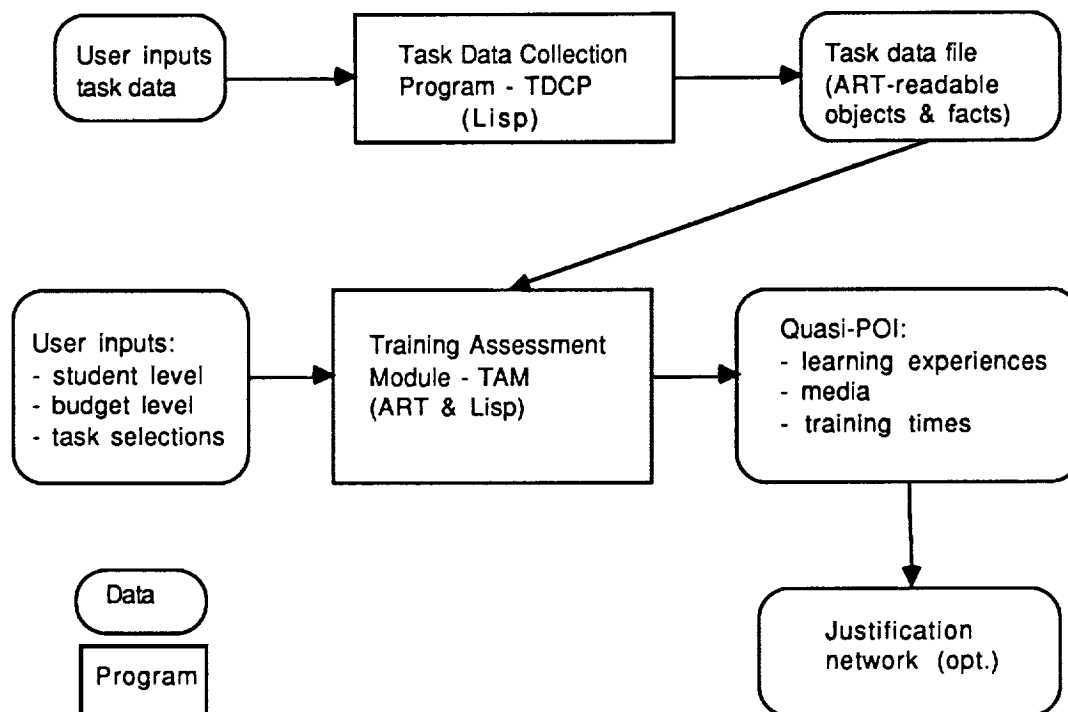


Fig. 1 TRAINING ASSESSMENT: CONTROL AND DATA FLOW

6.2 How to run the Task Data Collection Program (TDCP)

In the following description, some familiarity with the Symbolics environment and Genera 7.2 is assumed. User entries are italicized.

6.2.1 Loading TDCP

Note that Genera 7.2 is required to run the TDCP; it will not run correctly under Genera 7.1. If necessary, boot a Genera 7.2 world. Load TDCP from a Lisp Listener by entering:

Command: *Load File b:>carolyn>tas>ui>load-TDCP.lisp*

6.2.2 Running TDCP

Activate the menu-oriented task data collection program by invoking the function "tas-input":

Command: *(tas-input)*

This initiates the task data collection process by prompting for the name of a task. Enter the task name into a menu form, press <return>, and mouse click on "Do it".

Note that the "pop-up" menus have a similar method of operation: the user selects one or more items or types in a value, then mouses on "Do it" to enter the values into the program. "Abort" terminates the current data entry menu.

Once a task name has been entered, the menu of task characteristics appears. It is recommended to work through these attributes from top to bottom; all must be entered, except for TYPES, which is currently inactive. Selection of an attribute, such as CUES, brings up a menu of all currently allowable values for that attribute. Select any and all values which are salient task characteristics, then enter "Do it". For a more complete discussion of task attributes and values, please see the TASCs User's Guide. When all required values have been entered, select "DONE" from the menu.

The final information gathered about the current task is its degree of familiarity for each of a set of student types. (Currently the student types are listed in a variable, *student-list*, in the file "input-driver.lisp".) Three categories are available:

- (1) Old task, old equipment (no further training required)
- (2) Old task, new equipment
- (3) New task, new equipment

Specifying familiarity level concludes the characterization for the current task; the user is then prompted for a new task and the process is repeated. To terminate collection of task data, enter "nil" for the task name. Clicking on the Lisp listener window stops the program, which has formatted the task data into ART-readable facts and objects and written it to a task data file to be loaded with TAM.

6.3 How to run the Training Assessment Module (TAM)

A brief introduction to ART is given in the next section; for more information, the user is referred to ART Tutorials 1 and 3 and the ART Reference Manual.

6.3.1 Booting the ART world

If necessary, cold boot the machine with Genera 7.2 and ART as follows:

<select>-L to get to Lisp Listener

Command: *Halt Machine*

FEP: *boot art.boot* (this takes a few minutes)

Command: *Login <user-name>*

Press *<select>-A* to get to ART.

6.3.2 Getting around in ART

The left mouse button is used to select things like menu options, facts and rules from networks, etc. This usually has the effect of taking you down into the menu tree. The middle mouse button takes you back up; double middle (or shift middle) click with cursor arrow positioned on root menu window always brings you up all the way to the root menu, while a single middle click pops you up one level toward the root menu. (This description implies the menu space is viewed as an inverted tree, with the root menu at the top.)

Note that, if for some reason, ART does not respond to mouse clicks in ROOT menu, you can also enter root commands by typing them into the COMMAND window. (To enter commands into COMMAND window, position cursor over COMMAND window and click left before typing command.)

6.3.3 Loading TAM

First, click left on *clear* in ROOT window (in upper right corner of screen) to clear ART. Click left on *load*, then enter (in response to prompt in command window):

b:>carolyn>TAM>TAM-loadfile.lisp

Loading TAM takes a few minutes. Note that the task data is also loaded at this time. After the load process is finished, click left on *reset* to reset ART; this is required before each run.

6.3.4 Running TAM

There are a variety of options for viewing the actions of this KBS (Knowledge Based System) during the run; the following are simply suggestions. You may turn on viewing of FACTS and AGENDA by clicking on *watch*; click left on *facts* (should say YES) and *agenda* (should also say YES). Shape AGENDA window as prompted; suggested placement is in lower right section of screen. Note that watching various aspects of TAM as it runs slows down the run; this is especially true of the agenda.

The following steps set up and start a run. Go to the root menu by clicking shift-middle on root menu window; click left on *reset* if this has not been done; click left on *browse*, left on *assert*, and type *"/(TA)"*. ("TA" stands for Training Assessment.) This interaction will appear in the command window in upper left of screen and place the assertion (TA) into the fact base; this sets up a run of TAM. Obtain root menu in root menu window by clicking shift-middle in root menu window; then click left on *run*.

Select the incoming student training level and the budget level in response to the menu prompts. Then select the desired tasks from the task list menu and click on "Do it".

If agenda has been turned on during run, the agenda window shows rules being placed on the agenda when their conditions are satisfied. When rules come to top of agenda, they fire and place more assertions in the fact base. In this case, they are assigning learning experiences, media, and training times for the selected tasks.

At the conclusion of the run, an output window appears with the training requirements listed in tabular form. See Appendix D for a sample output display. For each task selected, a set of learning experiences, media, and training times appears; also, for each task, training times are summed over the learning experiences to produce a total training time for each task.

You can see how ART represents the task objects, or schemata as follows. Click left on *browse* in root menu window in upper right corner of screen, then click left on *schemata*, scroll till the desired task name appears, then click left on it. Click left on *text* to display ART code for the task. This shows the task characteristics, such as cues, systems, reasons for difficulty, task criticality, frequency of performance, etc., and their associated values.

6.3.5 Justification with ART

You can have TAM explain its reasoning by showing a justification network on a selected fact. A good way to access a specific fact in the fact base is to collect all facts having the desired relation; then select the desired fact from this subset. "Relation" refers to the first term in the fact. In the case of TAM, TRAINING-LE relations refer to assignment of learning experiences, TRAINING-MEDIUM relations to media, TRAINING-TIME-IDEAL relations to first approximation of training time, and TRAINING-TIME-ACTUAL relations to final computation of training time, with media taken into account.

To bring up a justification network, perform the following steps: From root menu, click left on *browse*, click left on *relations*, scroll to desired relation and click left. This brings up all the facts for the selected relation; click left on the desired fact to make it the current fact, and click left on *justification*. This brings up a justification network; if necessary, define window boundaries for the justification window. A sample justification network appears in Appendix D.

The justification network is a directed graph of interconnecting facts and rules. Each rule has on its left the facts which satisfied its conditions; arrows point from those facts to the rule. Any facts asserted by the rule are shown on the right with arrows pointing from the rule to the fact(s).

You can list facts which led to the current fact by clicking left on *justifiers*; also can show facts "downstream" from current fact by clicking on *justified*. Change the current fact by clicking on a different fact in the network. You can also show the text of a rule by clicking on the rule's box in the network; this selects it into root menu window. Then scroll till text option appears and click on it; shape window for text display if necessary.

7.0 APPENDICES

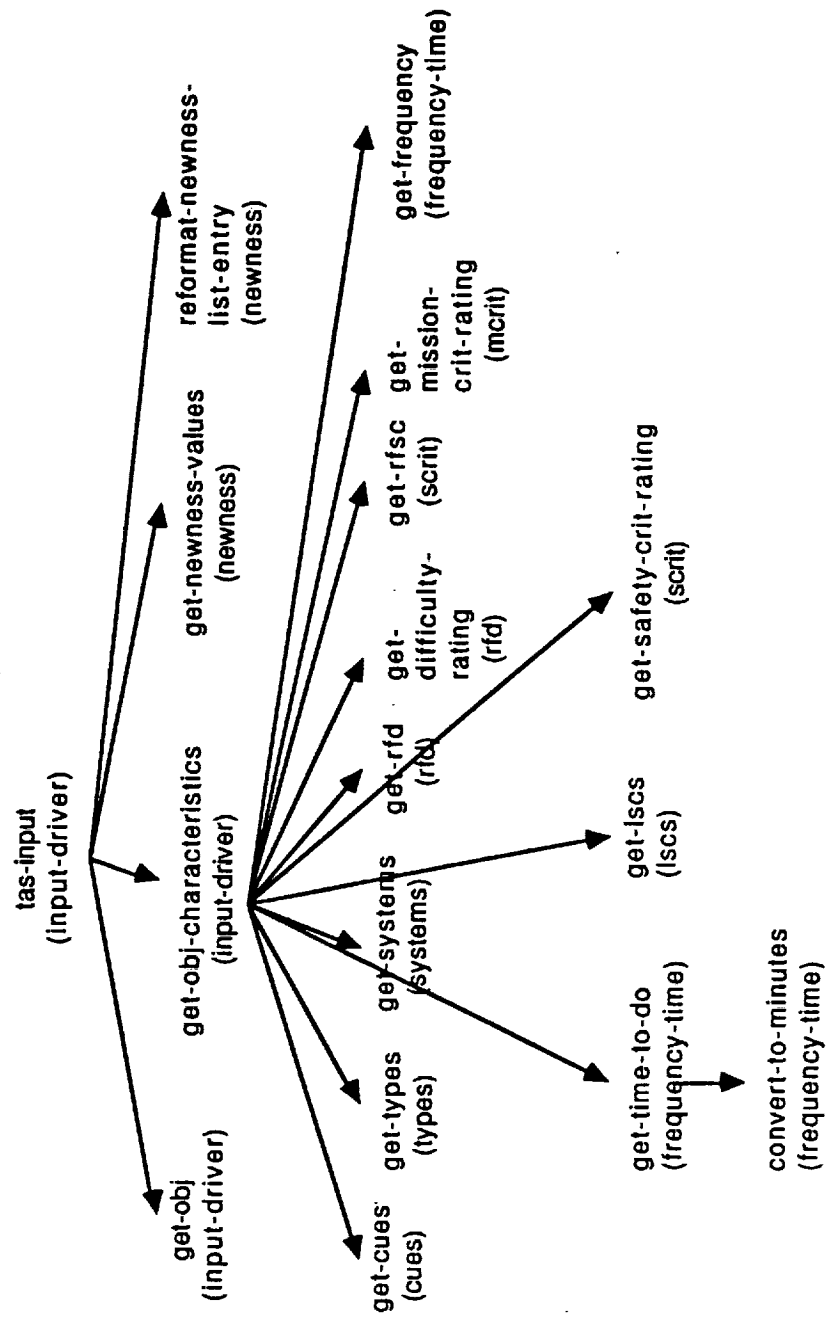
A. Glossary of Terms, Acronyms, or Abbreviations

ART	- Automated Reasoning Tool, an expert system building tool by Inference Corp.
HFE	- Human Factors Engineering
ISD	- Instructional Systems Design
KBS	- Knowledge Based System
MIDAS	- Man-machine Interface Design/Analysis System
POI	- Program Of Instruction
TAM	- Training Assessment Module
TASCS	- Training Analysis Support Computer System
TDCP	- Task Data Collection Program

B. Module Hierarchy Charts

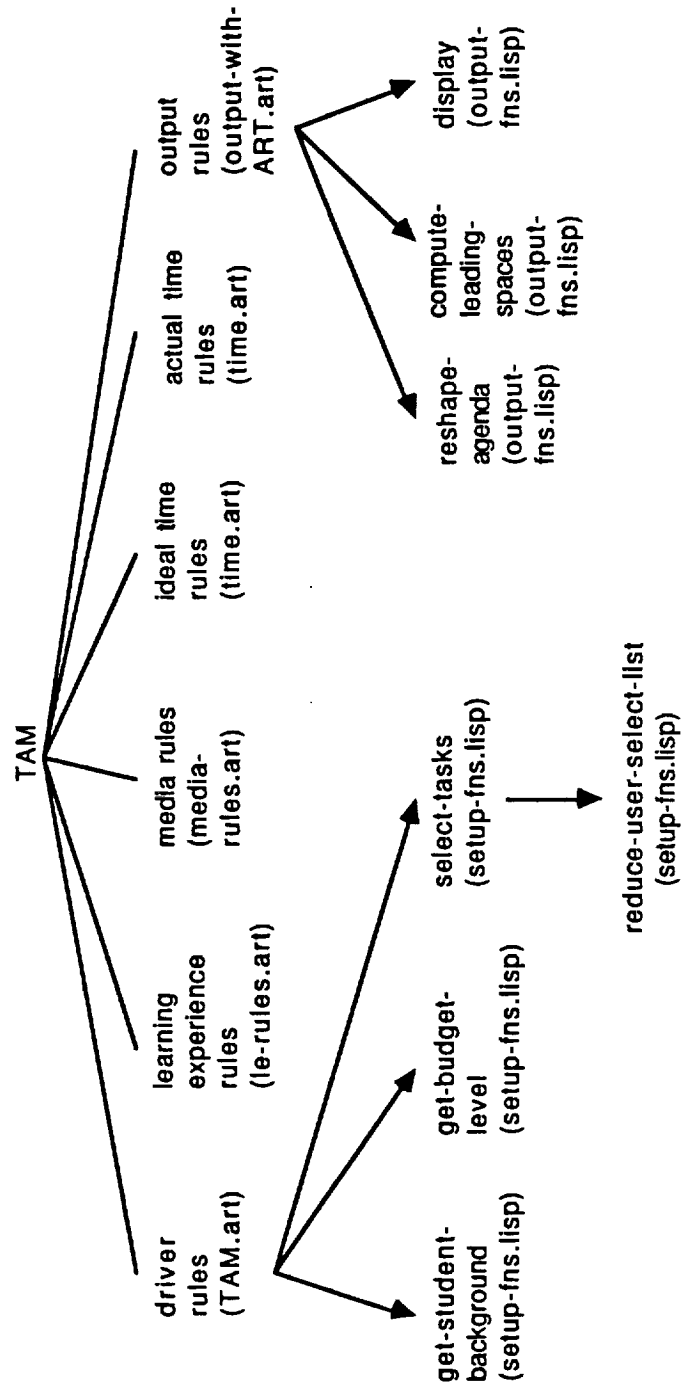
B.1 Task Data Collection Program (TDCP)

Appendix B.1: Module Hierarchy Chart, Task Data Collection Program



B.2 Training Assessment Module (TAM)

Appendix B.2: Module Hierarchy Chart, TAM (Training Assessment Module)



C. Sample Task Data File

BARRACUDA:>carolyn>tas>ui>obj-defschema.art.32

2/09/89 15:46:17 Page 1

;;; -- MODE: ART; BASE: 10.; PACKAGE: ART-USER --

```
(defschema SEND-RADIO-MESSAGE-WITHOUT-REMOTE
  (instance-of objective)
  (CUES TACTILE SOUND STILL-VISUAL)
  (TYPES)
  (SYSTEMS FLIGHT-CONTROLS COMMUNICATIONS)
  (RFD SIMULTANEOUS-TASKS DECISION-ON-SITUATION LITERAL-MEMORY OTHER-SHIP-OR-GROUND-PARTY-COORD)
  (DIFFICULTY-RATING 3)
  (RFSC NO-DEATH NO-INJURY NO-DAMAGE)
  (SAFETY-CRIT-RATING 1)
  (MISSION-CRIT-RATING 3)
  (FREQUENCY 4)
  (TIME-TO-DO 0.8333333)
  (LSCS VOICE-COMMUNICATION AURAL-DETECTION-CONSPICUOUS GROSS-MOTOR-1-OR-2-D
  USE-UNAIDED-PROCEDURE USE-UNAIDED-RULE REMEMBER-FACT)
)
```

```
(deffacts newness-SEND-RADIO-MESSAGE-WITHOUT-REMOTE
  (NEWNESS SEND-RADIO-MESSAGE-WITHOUT-REMOTE UHT O-N)
  (NEWNESS SEND-RADIO-MESSAGE-WITHOUT-REMOTE AH-64 O-O))
```

;;; -----

```
(defschema SEND-RADIO-MESSAGE-WITH-REMOTE
  (instance-of objective)
  (CUES TACTILE SOUND STILL-VISUAL)
  (TYPES)
  (SYSTEMS COMMUNICATIONS FLIGHT-CONTROLS)
  (RFD DECISION-ON-SITUATION LITERAL-MEMORY OTHER-SHIP-OR-GROUND-PARTY-COORD
  PRECISE-MANIPULATION)
  (DIFFICULTY-RATING 2)
  (RFSC NO-DEATH NO-INJURY NO-DAMAGE)
  (SAFETY-CRIT-RATING 1)
  (MISSION-CRIT-RATING 3)
  (FREQUENCY 4)
  (TIME-TO-DO 0.75)
  (LSCS VOICE-COMMUNICATION AURAL-DETECTION-CONSPICUOUS FINE-MOTOR-MULTI-D USE-UNAIDED-PROCEDURE
  USE-UNAIDED-RULE REMEMBER-FACT)
)
```

```
(deffacts newness-SEND-RADIO-MESSAGE-WITH-REMOTE
  (NEWNESS SEND-RADIO-MESSAGE-WITH-REMOTE UHT O-N)
  (NEWNESS SEND-RADIO-MESSAGE-WITH-REMOTE AH-64 O-O))
```

;;; -----

```
(defschema AUTO-TARGET-HANDOVER
  (instance-of objective)
  (CUES TACTILE SOUND DYNAMIC-VISUAL DISTANT-FOV WIDE-FOV)
  (TYPES)
  (SYSTEMS)
  (RFD ANXIETY OTHER-SHIP/GROUND-PARTY-COORD SIMULTANEOUS-TASKS ANTICIPATION LIMITED-TIME
  POSITION-ENERGY DECISION-ON-EQUIPMENT DECISION-ON-SITUATION MANY-CONTROLS)
  (DIFFICULTY-RATING 4)
  (RFSC DEATH-TO-SEVERAL)
  (SAFETY-CRIT-RATING 5)
  (MISSION-CRIT-RATING 5)
  (FREQUENCY 3)
  (TIME-TO-DO 1.0)
  (LSCS AURAL-DETECTION-CONSPICUOUS VISUAL-DETECTION-SUBTLE FINE-MOTOR-MULTI-D
  USE-UNAIDED-PRINCIPLE USE-AIDED-PROCEDURE REMEMBER-FACT)
)
```

```
(deffacts newness-AUTO-TARGET-HANDOVER
  (NEWNESS AUTO-TARGET-HANDOVER UHT N-N)
  (NEWNESS AUTO-TARGET-HANDOVER AH-64 O-O))
```

;;; -----

BARRACUDA:>carolyn>tas>ui>obj-defschema.art.32

2/09/89 15:46:17 Page 2

```

(defschema ENGAGE-TARGET-WITH-275-FFAR-NIGHT-6101
  (instance-of objective)
  (CUES TACTILE PHOTO-DETAIL WIDE-FOV COLOR)
  (TYPES)
  (SYSTEMS WEAPON-DELIVERY FLIGHT-CONTROLS)
  (RFD SIMULTANEOUS-TASKS TOO-MANY-CUES MANY-CONTROLS)
  (DIFFICULTY-RATING 4)
  (RFSC NO-DAMAGE)
  (SAFETY-CRIT-RATING 1)
  (MISSION-CRIT-RATING 4)
  (FREQUENCY 4)
  (TIME-TO-DO 1.0)
  (LSCS VISUAL-DETECTION-SUBTLE FINE-MOTOR-1-OR-2-D REMEMBER-FACT USE-UNAIDED-PROCEDURE
  USE-UNAIDED-RULE)
)

(deffacts newness-ENGAGE-TARGET-WITH-275-FFAR-NIGHT-6101
  (NEWNESS ENGAGE-TARGET-WITH-275-FFAR-NIGHT-6101 UHT N-N)
  (NEWNESS ENGAGE-TARGET-WITH-275-FFAR-NIGHT-6101 AH-64 O-O))

;;; -----

(defschema ENGAGE-TARGET-WITH-30-MM-GUN-NIGHT-6081
  (instance-of objective)
  (CUES TACTILE PHOTO-DETAIL WIDE-FOV COLOR)
  (TYPES)
  (SYSTEMS WEAPON-DELIVERY FLIGHT-CONTROLS)
  (RFD SIMULTANEOUS-TASKS TOO-MANY-CUES MANY-CONTROLS)
  (DIFFICULTY-RATING 4)
  (RFSC NO-DAMAGE)
  (SAFETY-CRIT-RATING 1)
  (MISSION-CRIT-RATING 4)
  (FREQUENCY 4)
  (TIME-TO-DO 0.6666667)
  (LSCS VISUAL-DETECTION-SUBTLE FINE-MOTOR-1-OR-2-D REMEMBER-FACT USE-UNAIDED-PROCEDURE
  USE-UNAIDED-RULE)
)

(deffacts newness-ENGAGE-TARGET-WITH-30-MM-GUN-NIGHT-6081
  (NEWNESS ENGAGE-TARGET-WITH-30-MM-GUN-NIGHT-6081 UHT N-N)
  (NEWNESS ENGAGE-TARGET-WITH-30-MM-GUN-NIGHT-6081 AH-64 O-O))

;;; -----

(defschema ENGAGE-TARGET-WITH-HELLFIRE-NIGHT-6080
  (instance-of objective)
  (CUES TACTILE PHOTO-DETAIL WIDE-FOV COLOR)
  (TYPES)
  (SYSTEMS WEAPON-DELIVERY FLIGHT-CONTROLS)
  (RFD ANXIETY SYSTEM-DESIGN SIMULTANEOUS-TASKS DECISION-ON-EQUIPMENT TOO-MANY-CUES
  MANY-CONTROLS POSITION-ENERGY PRECISE-MANIPULATION)
  (DIFFICULTY-RATING 5)
  (RFSC NO-DAMAGE)
  (SAFETY-CRIT-RATING 1)
  (MISSION-CRIT-RATING 4)
  (FREQUENCY 4)
  (TIME-TO-DO 1.25)
  (LSCS VISUAL-DETECTION-SUBTLE FINE-MOTOR-1-OR-2-D USE-UNAIDED-PROCEDURE USE-UNAIDED-RULE
  REMEMBER-CONCEPT)
)

(deffacts newness-ENGAGE-TARGET-WITH-HELLFIRE-NIGHT-6080
  (NEWNESS ENGAGE-TARGET-WITH-HELLFIRE-NIGHT-6080 UHT N-N)
  (NEWNESS ENGAGE-TARGET-WITH-HELLFIRE-NIGHT-6080 AH-64 O-O))

;;; -----

(defschema PERFORM-ECU-LOCKOUT-OPS-4007
  (instance-of objective)
  (CUES COLOR DYNAMIC-VISUAL)
  (TYPES)
  (SYSTEMS FLIGHT-CONTROLS FLIGHT-INSTRUMENTS ENGINE-SYSTEMS)
  (RFD SPECIFIC-SEQUENCE DECISION-ON-EQUIPMENT)
)

```

```
(DIFFICULTY-RATING 2)
(RFSC MAJOR-DAMAGE)
(SAFETY-CRIT-RATING 3)
(MISSION-CRIT-RATING 3)
(FREQUENCY 1)
(TIME-TO-DO 0.25)
(LSCS VISUAL-DETECTION-CONSPICUOUS REMEMBER-PROCEDURE)
)

(deffacts newness-PERFORM-ECU-LOCKOUT-OPS-4007
  (NEWNESS PERFORM-ECU-LOCKOUT-OPS-4007 UHT N-N)
  (NEWNESS PERFORM-ECU-LOCKOUT-OPS-4007 AH-64 O-N))

;;; -----

(defschema PERFORM-OGE-HOVER-WITH-SINGLE-ENG-FAILURE-4001
  (instance-of objective)
  (CUES DISTANT-FOV WIDE-FOV DYNAMIC-VISUAL)
  (TYPES)
  (SYSTEMS ENGINE-SYSTEMS FLIGHT-CONTROLS)
  (RFD ANXIETY DECISION-ON-EQUIPMENT PRECISE-MANIPULATION POSITION-ENERGY)
  (DIFFICULTY-RATING 4)
  (RFSC NO-INJURY NO-DAMAGE)
  (SAFETY-CRIT-RATING 1)
  (MISSION-CRIT-RATING 2)
  (FREQUENCY 2)
  (TIME-TO-DO 0.75)
  (LSCS VISUAL-DETECTION-CONSPICUOUS FINE-MOTOR-1-OR-2-D USE-UNAIDED-PRINCIPLE REMEMBER-FACT)
)

(deffacts newness-PERFORM-OGE-HOVER-WITH-SINGLE-ENG-FAILURE-4001
  (NEWNESS PERFORM-OGE-HOVER-WITH-SINGLE-ENG-FAILURE-4001 UHT N-N)
  (NEWNESS PERFORM-OGE-HOVER-WITH-SINGLE-ENG-FAILURE-4001 AH-64 O-N))

;;; -----

(defschema PERFORM-SLOPE-OPS-VMC-3511
  (instance-of objective)
  (CUES MOTION DISTANT-FOV WIDE-FOV DYNAMIC-VISUAL)
  (TYPES)
  (SYSTEMS POWER-TRAIN-SYSTEM FLIGHT-CONTROLS)
  (RFD VAGUE-CUES PRECISE-MANIPULATION POSITION-ENERGY)
  (DIFFICULTY-RATING 3)
  (RFSC MINOR-INJURY MAJOR-DAMAGE)
  (SAFETY-CRIT-RATING 3)
  (MISSION-CRIT-RATING 4)
  (FREQUENCY 4)
  (TIME-TO-DO 0.41666666)
  (LSCS VISUAL-DETECTION-CONSPICUOUS FINE-MOTOR-MULTI-D REMEMBER-RULE)
)

(deffacts newness-PERFORM-SLOPE-OPS-VMC-3511
  (NEWNESS PERFORM-SLOPE-OPS-VMC-3511 UHT O-N)
  (NEWNESS PERFORM-SLOPE-OPS-VMC-3511 AH-64 O-N))

;;; -----

(defschema START-ENGINE-WITH-PRESS-AIR-SOURCE-1607
  (instance-of objective)
  (CUES SOUND DYNAMIC-VISUAL)
  (TYPES)
  (SYSTEMS ENGINE-SYSTEMS)
  (RFD DECISION-ON-EQUIPMENT)
  (DIFFICULTY-RATING 1)
  (RFSC MINOR-DAMAGE)
  (SAFETY-CRIT-RATING 2)
  (MISSION-CRIT-RATING 2)
  (FREQUENCY 3)
  (TIME-TO-DO 0.25)
  (LSCS AURAL-DETECTION-SUBTLE VISUAL-DETECTION-CONSPICUOUS USE-AIDED-PROCEDURE)
)

(deffacts newness-START-ENGINE-WITH-PRESS-AIR-SOURCE-1607
```

BARRACUDA:>carolyn>tas>ui>obj-defschema.art.32

2/09/89 15:46:17 Page 4

(NEWNESS START-ENGINE-WITH-PRESS-AIR-SOURCE-1607 UHT O-N)
(NEWNESS START-ENGINE-WITH-PRESS-AIR-SOURCE-1607 AH-64 O-O))

::: -----

D. Sample Displays

D.1 Task Attribute Input Displays from TDCP

Please enter task name
Task name: SEND-RADIO-MESSAGE-WITH-REMOTE
Exit <input type="checkbox"/>

FOR TASK SELECT-RADIO, CHOOSE ONE CHARACTERISTIC TO ENTER NEXT
<p>CUES TYPES SYSTEMS REASONS FOR DIFFICULTY DIFFICULTY RATING REASONS FOR SAFETY CRITICALITY MISSION CRITICALITY RATING FREQUENCY TIME TO DO LEARNING SUBCATEGORIES</p> <p>done</p>

CUES FOR TASK: SEND-RADIO-MESSAGE-WITH-REMOTE	
Do it	None
Still Visual	Dynamic Visual
Wide Field of View	Distant (Out the Window) Field of View
Color	Photographic Detail
Motion	Sound
Tactile	

Select SYSTEMS for task: SEND-RADIO-MESSAGE-WITH-REMOTE
Do it
Exit
1. Helicopter
Emergency equipment
Engine systems
Fuel system
Flight controls
Hydraulic systems
Power train system
Environmental control systems
Power supply and distribution system
Lighting
Flight instruments
2. Avionics
Communications
Navigation
Sensors
3. Mission Equipment
Weapon delivery
Electronic counter measures

D.1 Task Attribute Input Displays from TDCP, cont.

REASONS FOR DIFFICULTY FOR TASK: SEND-RADIO-MESSAGE-WITH-REMOTE	
Do it	None
1. PSYCHOMOTOR	
Exertion	
Appreciation of Position and Energy	
Precise Manipulation	
Many Controls	
2. CUES	
Too Many Cues	
Few Cues	
Distracting Cues	
Vague Cues	
3. COGNITION	
Decision on Mission	
Decision on Situation	
Decision on Equipment	
Conceptual Memory	
Literal Memory	
4. TIME SEQUENCE	
Limited Time	
Lengthy	
Anticipation	
Specific Sequence	
Simultaneous Tasks	
5. MISCELLANEOUS	
Other Ship/Ground Party Coordination	
Unfamiliarity	
Anxiety	
Engineering System Design	

Difficulty Rating for SEND-RADIO-MESSAGE-WITH-REMOTE	
Difficulty Rating: 1 2 3 4 5	
Exit <input type="checkbox"/>	

REASONS FOR SAFETY CRITICALITY for task: SEND-RADIO-MESSAGE-WITH-REMOTE			
Do it	None		
DAMAGE	INJURY	DEATH	(RATING)
No Damage	No Injury	No Death	1
Minor Damage	Minor Injury		2
Major Damage	Major Injury		3
Catastrophic Damage	Maj Injury to Several People	Death to 1 person	4
		Death to several people	5

Enter MISSION CRITICALITY rating for task: SEND-RADIO-MESSAGE-WITH-REMOTE	
Do it	None
CATEGORY	CRITICALITY RATING
No mission effect	1
Minor added work	2
Major added work	3
Mission partially ineffective	4
Mission non-effective	5

D.1 Task Attribute Input Displays from TDCP, cont.

Estimate time to perform task SEND-RADIO-MESSAGE-WITH-REMOTE	
Hours :	00
Minutes:	00
Seconds:	45
Exit <input type="checkbox"/>	

Estimate FREQUENCY of performance for task SEND-RADIO-MESSAGE-WITH-REMOTE	
An occasional flight (or less)	
Every few flights	
Once per flight	
Several times per flight	
All during the flight	

LEARNING SUBCATEGORIES (COGNITIVE) for task: SEND-RADIO-MESSAGE-WITH-REMOTE				
Do it	None			
Remember fact	Remember concept	Remember procedure	Remember rule	Remember principle
	Use aided concept	Use aided procedure	Use aided rule	Use aided principle
	Use unaided concept	Use unaided procedure	Use unaided rule	Use unaided principle

LEARNING SUBCATEGORIES (PSYCHOMOTOR) for task: SEND-RADIO-MESSAGE-WITH-REMOTE	
Do it	None
Gross motor - one or two dimensions	Gross motor - multi-dimensional
Fine motor - one or two dimensions	Fine motor - multi-dimensional

LEARNING SUBCATEGORIES (MISCELLANEOUS) for task: SEND-RADIO-MESSAGE-WITH-REMOTE	
Do it	None
Visual detection - Conspicuous	
Visual detection - Subtle	
Aural detection - Conspicuous	
Aural detection - Subtle	
Voice communication	
Attitudinal	

FOR TASK SEND-RADIO-MESSAGE-WITH-REMOTE AND STUDENT AH-64, INDICATE DEGREE OF FAMILIARITY	
Old task, old equipment	
Old task, new equipment	
New task, new equipment	

FOR TASK SEND-RADIO-MESSAGE-WITH-REMOTE AND STUDENT UH1, INDICATE DEGREE OF FAMILIARITY	
Old task, old equipment	
Old task, new equipment	
New task, new equipment	

D.2 Sample task object with characteristics

<div>COMMAND WINDOW</div> <div>SEND-RADIO-MESSAGE-WITH-REMOTE</div> <div><div>edit</div><div>text</div><div>facts</div><div>relational network</div><div>inheritance network</div><div>related actions</div><div>slots</div></div>	<div>SEND-RADIO-MESSAGE-WITH-REMOTE</div>
TRAINING ASSESSMENT QUASI-POI	
<div>Schema SEND-RADIO-MESSAGE-WITH-REMOTE</div> <div>(DEFSCHEMA SEND-RADIO-MESSAGE-WITH-REMOTE</div> <div>(INSTANCE-OF OBJECTIVE)</div> <div>(CUES HOSTILE)</div> <div>(CUES SOUND)</div> <div>(CUES STILL-VISUAL)</div> <div>(TYPES)</div> <div>(SYSTEMS COMMUNICATIONS)</div> <div>(SYSTEMS FLIGHT-CONTROLS)</div> <div>(RFID DECISION-ON-SITUATION)</div> <div>(RFID LITTERAL-MEMORY)</div> <div>(RFID OTHER-SHIP-OR-GROUND-PARTY-COORD)</div> <div>(RFID PRECISE-MANIPULATION)</div> <div>(DIFFICULTY-RATING 2)</div> <div>(RFSC NO-DEATH)</div> <div>(RFSC NO-INJURY)</div> <div>(RFSC NO-DAMAGE)</div> <div>(SAFETY-CRIT-RATING 1)</div> <div>(MISSION-CRIT-RATING 3)</div> <div>(FREQUENCY 4)</div> <div>(TIME-TO-DO 0.75)</div> <div>(LSCS VOICE-COMMUNICATION)</div> <div>(LSCS MORAL-DETECTION-CONSPICUOUS)</div> <div>(LSCS FINE-MOTOR-MULTI-D)</div> <div>(LSCS USE-UNRAIDED-PROCEDURE)</div> <div>(LSCS USE-UNRAIDED-RULE)</div> <div>(LSCS REMEMBER-FACT)</div> <div>)</div>	<div>MEDIUM</div> <div>INTERAC</div> <div>DFT</div> <div>DFT</div> <div>AGENDA</div> <div>1000000 COLLECT-TASKS (P-1368,)</div> <div>1000000 COLLECT-TASKS (P-1347,)</div> <div>1000000 COLLECT-TASKS (P-1325,)</div> <div>1000000 COLLECT-TASKS (P-1308,)</div> <div>1000000 COLLECT-TASKS (P-1279,)</div> <div>1000000 COLLECT-TASKS (P-1255,)</div> <div>1000000 COLLECT-TASKS (P-1231,)</div> <div>1000000 COLLECT-TASKS (P-1202,)</div> <div>1000000 COLLECT-TASKS (P-1177,)</div> <div>1000000 COLLECT-TASKS (P-1152,)</div>
<div>(Thu 16 Feb 8:12:56) carolyn CL ACU: User Input Spid</div>	

D.3 Sample rules

<pre> (DEFRULE LE-DEMO4 (FIND-TRAINING 700BJECTIVE) (JOIN (DIFFICULTY-RATING 700BJECTIVE (DIFFICULTY-RATING: (NL)>= 700BJECTIVE-RATING 4)) (RFD 700BJECTIVE (NL)?parameter-temp-1 & PRECISE-MANIPULATION POSITION-ENERGY OTHER-SHIP-OR-GROUND-PARTY-COORD UNFAMILIARITY)) => (ASSERT (TRAINING-LE 700BJECTIVE DEMONSTRATION))) </pre>	<p>LE-DEMO4</p> <p>watched: NO breakpoint: NO refresh undefrule salience: 0 referenced relations asserted relations text display net</p>
<pre> (DEFRULE LE-DEMO5 (FIND-TRAINING 700BJECTIVE) (JOIN (DIFFICULTY-RATING 700BJECTIVE (DIFFICULTY-RATING: (NL)>= 700BJECTIVE-RATING 4)) (RFD 700BJECTIVE LIMITED-TIME) (RFD 700BJECTIVE ANTICIPATION)) => (ASSERT (TRAINING-LE 700BJECTIVE DEMONSTRATION))) </pre>	<p>LE-DEMOS</p> <p>PERIENCE MEDIUM</p> <p>INTERRC OFI OFI</p>
<pre> (DEFRULE LE-DEMO6 (FIND-TRAINING 700BJECTIVE) (JOIN (DIFFICULTY-RATING 700BJECTIVE (DIFFICULTY-RATING: (NL)>= 700BJECTIVE-RATING 4)) (LSDS 700BJECTIVE (NL)?parameter-temp-1 & USE-ALIBED-CONCEPT USE-UNRAIDED-CONCEPT) (LSDS 700BJECTIVE (NL)?parameter-temp-2 & USE-ALIBED-PROCEDURE USE-UNRAIDED-PROCEDURE)) => (ASSERT (TRAINING-LE 700BJECTIVE DEMONSTRATION))) </pre>	<p>LE-DEMO6</p> <p>AGENDA</p> <p>1000000 COLLECT-TASKS (F-1358.) 1000000 COLLECT-TASKS (F-1347.) 1000000 COLLECT-TASKS (F-1325.) 1000000 COLLECT-TASKS (F-1308.) 1000000 COLLECT-TASKS (F-1279.) 1000000 COLLECT-TASKS (F-1255.) 1000000 COLLECT-TASKS (F-1231.) 1000000 COLLECT-TASKS (F-1202.) 1000000 COLLECT-TASKS (F-1177.) 1000000 COLLECT-TASKS (F-1152.)</p>

[11w 16 Feb 8:20:26] carolyn

CL ACU:

User Input

Solid

D.3 Sample rules, cont.

<p>COMMAND WINDOW</p> <pre> => text => => text => pop => MEDIA-EXPL-LECTURE-VIS-AID </pre>	<p>MEDIA-EXPL-LECTURE-VIS-AID</p> <pre> watched: NO breakpoint: NO refresh underrule salience: -1 referenced relations asserted relations text display net </pre>
<p>TRAINING ASSESSMENT QURSI-POI</p> <pre> MEDIA-EXPL-LECTURE-VIS-AID (DEFRULE MEDIA-EXPL-LECTURE-VIS-AID (DECLARE (SALIENCE ?*MEDIA-LEVEL1-SALIENCE)) (TRAINING-LE ?OBJECTIVE ?LE&TEXTUAL-EXPLANATION) GRAPHIC-EXPLANATION) DYNAMIC-EXPLANATION) (JOIN (FREQUENCY ?OBJECTIVE ?FREQUENCY&:(NL)= ?FREQUENCY 4)) (MISSION-CRIT-RATING ?OBJECTIVE ?MCRT-RATING&: (NL)= ?MCRT-RATING 4)) (SAFETY-CRIT-RATING ?OBJECTIVE ?SCRT-RATING&: (NL)= ?SCRT-RATING 4))) => (ASSERT (TRAINING-MEDIUM ?OBJECTIVE ?LE LECTURE-WITH-VISUAL-AIDS))) </pre>	<pre> MEDIA-EXPL-VIDEOODISC-CBT (DEFRULE MEDIA-EXPL-VIDEOODISC-CBT (DECLARE (SALIENCE ?*MEDIA-LEVEL3-SALIENCE)) (TRAINING-LE ?OBJECTIVE ?LE&DYNAMIC-EXPLANATION) (NOT (TRAINING-MEDIUM ?OBJECTIVE ?LE LECTURE-WITH-VISUAL-AIDS)) (NOT (TRAINING-MEDIUM ?OBJECTIVE ?LE VIDEOTAPE))) => (ASSERT (TRAINING-MEDIUM ?OBJECTIVE ?LE VIDEOODISC-CBT))) </pre> <pre> MEDIA-COG-PSY-PTT-DEDICATED-PTT (DEFRULE MEDIA-COG-PSY-PTT-DEDICATED-PTT (DECLARE (SALIENCE ?*MEDIA-LEVEL1-SALIENCE*)) (TRAINING-LE ?OBJECTIVE ?LE&COGNITIVE-PTT(PSYCHOMOTOR-PTT) (JOIN (FREQUENCY ?OBJECTIVE ?FREQUENCY&:(NL)= ?FREQUENCY 4)) (MISSION-CRIT-RATING ?OBJECTIVE ?MCRT-RATING&: (NL)= ?MCRT-RATING 3))) (BUDGET ?NL)?parameter-temp-1 A^LOW) => (ASSERT (TRAINING-MEDIUM ?OBJECTIVE ?LE DEDICATED-PTT))) </pre>

D.4 Sample agenda

COMMAND WINDOW		ROOT	
<pre> => run ==> f-1406 [STUDENT UNIT] ==> f-1407 [BUDGET HIGH] ==> f-1408 [TASK-LIST [AUTO-TARGET-HANDOVER SEND-RADIO-MESSAGE-WITHOUT-REMOIE]] ENOTE SEND-RADIO-MESSAGE-WITHOUT-REMOIE]] ==> f-1409 [FIND-TRAINING AUTO-TARGET-HANDOVER] ==> f-1410 [FIND-TRAINING SEND-RADIO-MESSAGE-WITHOUT-REMOIE] ==> f-1411 [FIND-TRAINING SEND-RADIO-MESSAGE-WITHOUT-REMOIE] ==> f-1412 [TRAINING-LE AUTO-TARGET-HANDOVER DYNAMIC-EXPLANATION] Program halted. </pre>		<pre> clear load reset watch run step browse icon editor miscellaneous examples exit </pre>	
<p>AGENDA</p> <pre> 200 LE-EXPLANATION-DYNAMIC1 (f-1409,f-1206) 100 LE-EXPLANATION-GRAPHIC1 (f-1411,f-1156,,) 0 LE-FT11-disJunct-3 (f-1411,f-1168) 0 LE-FT11-disJunct-2 (f-1411,f-1167) 0 LE-FT11-disJunct-1 (f-1411,f-1163) 0 LE-FT11-disJunct-3 (f-1418,f-1193) 0 LE-FT11-disJunct-2 (f-1418,f-1192) 0 LE-FT11-disJunct-1 (f-1418,f-1188) 0 LE-DEM02-disJunct-2-disJunct-2 (f-1418,f-1187)) 0 LE-DEM02-disJunct-2-disJunct-1 (f-1418,f-1198)) 0 LE-FT11-disJunct-3 (f-1409,f-1222) 0 LE-FT11-disJunct-2 (f-1409,f-1221) 0 LE-FT11-disJunct-1 (f-1409,f-1219) 0 LE-AT1111UDINHL-PT115 (f-1409,f-1222,f-1406,f-1398) 0 LE-PSYCHOM0100-PT111-disJunct-1 (f-1409,f-1219,f-1227,f-1406,f-1398) 0 LE-COGNITIVE-PT11-disJunct-4 (f-1409,f-1219,f-1226,f-1406,f-1398) 0 LE-COGNITIVE-PT11-disJunct-2 (f-1409,f-1219,f-1228,f-1406,f-1398) 0 LE-COGNITIVE-PT11-disJunct-1 (f-1409,f-1219,f-1217),f-1406,f-1398) 0 LE-COGNITIVE-PT11-disJunct-1 (f-1409,f-1219,f-1216),f-1406,f-1398) 0 LE-DEM05 (f-1409,f-1219,f-1214,f-1213) 0 LE-DEM04 (f-1409,f-1219,f-1215) 0 LE-DEM03 (f-1409,f-1219,f-1211) 0 LE-DEM02-disJunct-2-disJunct-1 (f-1409,f-1222,f-1227)) 0 LE-DEM02-disJunct-1-disJunct-1 (f-1409,f-1219,f-1227)) 0 LE-DEM01 (f-1409,f-1221) -1 MEDIA-EXPL-LECTURE-VIS-RID (f-1412,(f-1223,f-1222,f-1221)) -3 MEDIA-EXPL-VIDEOBISC-CBI (f-1412,,) </pre>			
<p>L: 200 LE-EXPLANATION-DYNAMIC1 (f-1409,f-1206) L2: Window menu, M: Pop. M2: Root. R: Help for this option.</p>			
<p>[Thu 16 Feb 8:32:24] carolyn</p>		<p>User Input</p>	

D.5 Output screen

ROOT

clear
load
reset
watch
run

SKEP
browse
icon editor
miscellaneous

COMMAND WINDOW
OUT-REMOTE FULL-TASK-TRAINING OFT 10
<== F-1502 [PRINT-MARKED-RESULTS SEND-RADIO-MESSAGE
-WITHOUT-REMOTE]
No applicable rules.

=>
OUT-REMOTE GRAPHIC-EXPLANATION INTERACTIVE-SLIDE-TAP
E 15
<== F-1508 [PRINT-TRAINING SEND-RADIO-MESSAGE-WITH

TRAINING ASSESSMENT QUASI-POI

INCOMING STUDENT: UHT
BUDGET LEVEL: HIGH

LEARNING OBJECTIVE

LEARNING EXPERIENCE

MEDIUM

TIME TO TRAIN (MINUTES)

COST (APPROX.)

AUTO TARGET HANDOVER

DYNAMIC EXPLANATION
DEMONSTRATION
COGNITIVE PTT
PSYCHOMOTOR PTT
AFFECTIVE PTT
FULL TASK TRAINING

LECTURE WITH VISUAL AIDS
UHT
DEDICATED PTT
DEDICATED PTT
ACTUAL SYSTEM
UHT

194
44
192
240
50
150

780 (13.0 HRS)

SEND RADIO MESSAGE WITH REMOTE

GRAPHIC EXPLANATION
DEMONSTRATION
FULL TASK TRAINING

INTERACTIVE SLIDE TAPE
OFT
OFT

15
4
9

28 (0.5 HRS)

SEND RADIO MESSAGE WITHOUT REMOTE

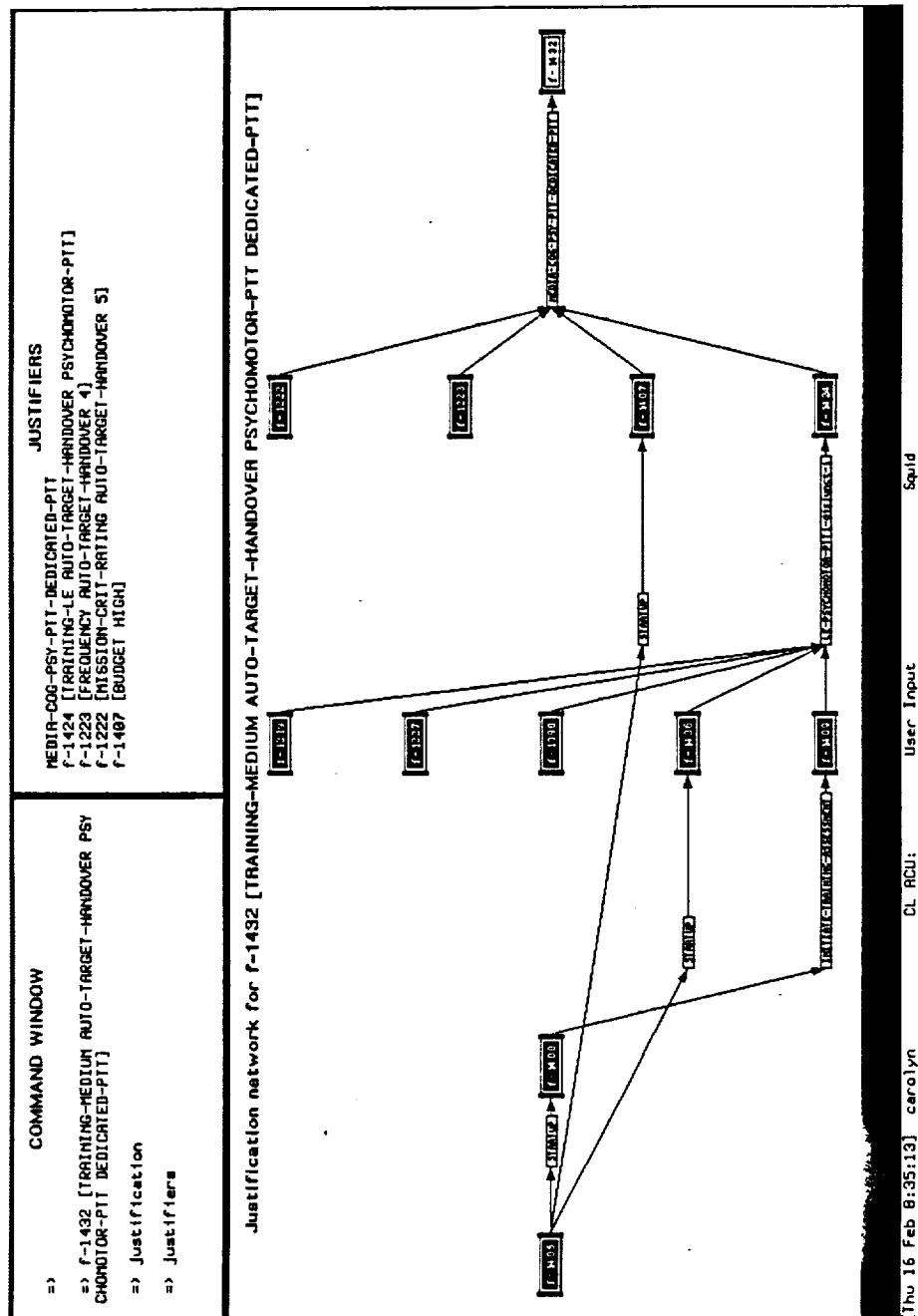
GRAPHIC EXPLANATION
FULL TASK TRAINING

INTERACTIVE SLIDE TAPE
OFT

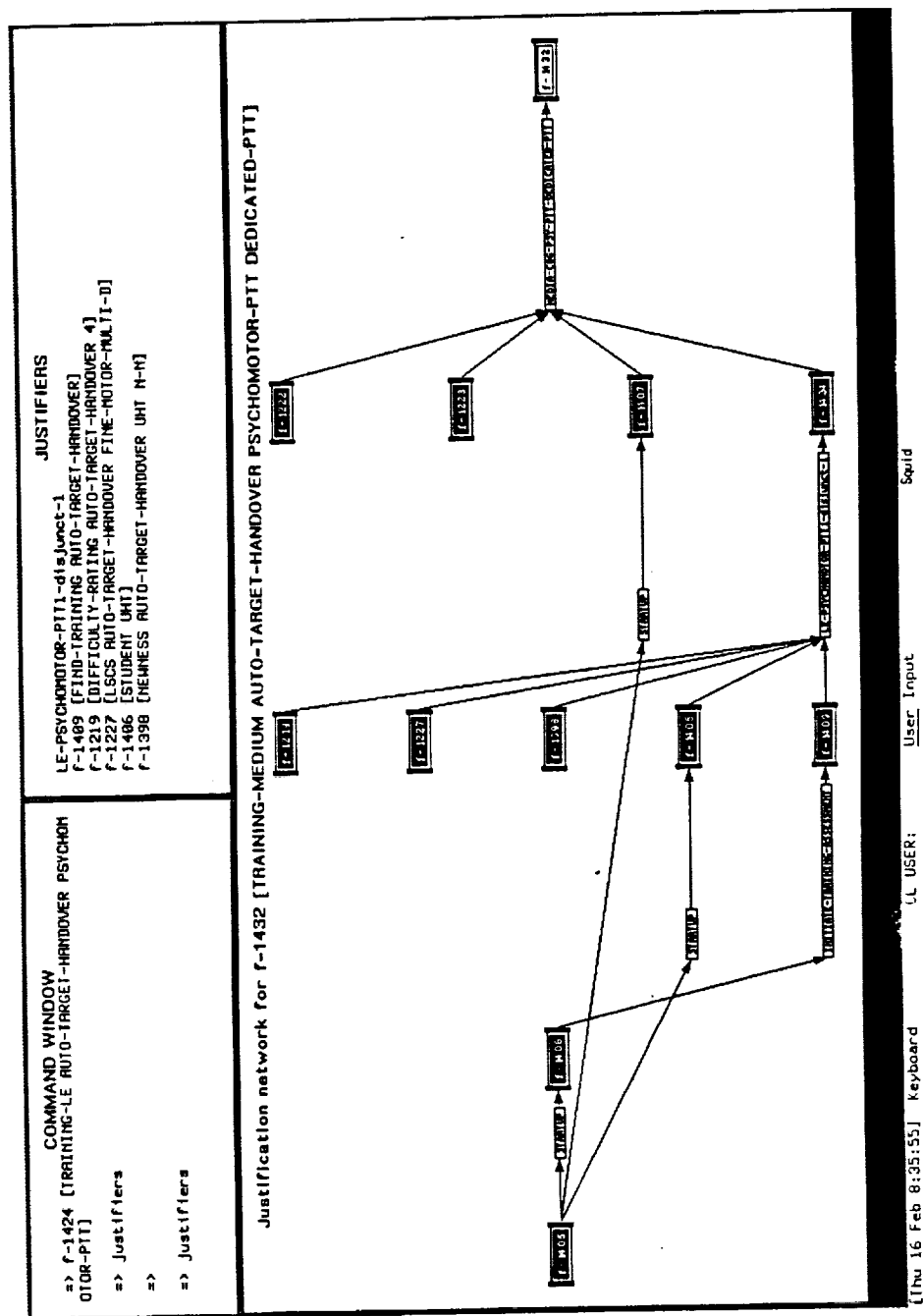
15
10

25 (0.4 HRS)

D.6 Justification networks



D.6 Justification networks, cont.



Annex F

Army-NASA Aircrew/Aircraft Integration Program

A³I

**Software Detailed Design Document:
Aero/Guidance**

prepared by

Alex Chiu

December 1988

Table of Contents

1.0 INTRODUCTION.....	F-1
1.1 Identification	F-1
1.2 Scope.....	F-1
1.3 Purpose	F-1
2.0 RELATED DOCUMENTATION.....	F-1
2.1 Applicable Documents	F-1
2.2 Information Documents	F-2
3.0 REQUIREMENTS AND DESIGN APPROACH.....	F-2
3.1 Requirements and Rationale.....	F-2
3.2 Hardware Environment	F-4
3.3 Software Environment	F-4
4.0 DETAILED DESIGN DESCRIPTION	F-5
4.1 Organization.....	F-5
4.2 Unit Detailed Design	F-7
4.2.1 openfiles.f.....	F-7
4.2.2 tdblkdatx.f	F-7
4.2.3 tinput.f.....	F-7
4.2.4 fly.f.....	F-8
4.2.5 tguidap2x.f.....	F-8
4.2.6 newpsix.f	F-8
4.2.7 tmangx.f.....	F-9
4.2.8 imwind.c.....	F-9
4.2.9 animate.c.....	F-9
4.2.10 simdata.c.....	F-9
4.2.11 path.c.....	F-9
4.2.12 mgcom.c.....	F-10
4.2.13 mgiedit.c.....	F-10
4.2.14 checkerboard.2.....	F-10
4.2.15 waypoints.....	F-10
4.2.16 animateflt.doc	F-10
4.2.17 animate.menu	F-11
4.2.18 message.h	F-11
4.2.19 message.common.....	F-11
4.2.20 heli.h	F-11
4.2.21 heli.common	F-11
5.0 NOTES	F-12
5.1 Miscellaneous	F-12
5.2 Limitations.....	F-12
5.3 Future Directions	F-12
6.0 USERS GUIDE	F-12

1.0 INTRODUCTION

1.1 Identification

This document establishes the requirements and detailed design of the Flight Dynamics/Guidance Computer Software Configuration Item (CSCI), which forms a part of the A3I Computer Program System. Descriptions of the detailed processing requirements, structure, I/O, and control are provided for each lower level Computer Software Component (CSC), unit, or function contained within the CSCI.

1.2 Scope

This document is primarily focused on the Phase III capabilities of the flight dynamics/guidance CSCI -- particularly the integration of the Analytical Mechanics Associates (AMA) developed Fortran code -- TGUIDAP2/TMAN with the A3I Views CSCI. Detailed descriptions of the AMA developed code is not included herein. Rather, the reader is referred to the AMA Report 252-3 listed in Section 2.1 -- *A3I Autopilot/Guidance Program Homing/Path Guidance with Turn-Straight-Turn Option*. It is assumed that the reader is familiar with the simple concepts of aerodynamics, the basic theory of aircraft stability and control, and UNIX, Fortran, and C.

1.3 Purpose

The purpose of the Flight Dynamics/Guidance CSCI is to provide representative models of a generalized helicopter aerodynamics quantities and guidance capabilities for use in the A3I simulation. Previous development phases contained simplified, but somewhat inadequate models of these functions for A3I's purposes. Analytical Mechanics Associates has developed a well-tested and accepted Fortran model of helicopter dynamics and guidance previously used on VAX computers. This model is linear and partly decoupled, in the sense that the collective control has no effect on the yaw, pitch, or roll movement, and without taking into consideration the wind effect on the helicopter behavior. This model has been improved with the addition of a Turn-Straight-Turn guidance scheme and ported to the A3I's Silicon Graphics Workstations. The Flight Dynamics/Guidance CSCI also serves as a method to represent to the Symbolics pilot model the control inputs and their durations needed to maneuver the simulated craft to the desired waypoints contained in the mission.

2.0 RELATED DOCUMENTATION

2.1 Applicable Documents

A. Gessow, G. C. Myers, Jr., *Aerodynamics of the Helicopter*, Frederick Ungar Publishing Co., New York, December, 1952

M. S. Lewis, E. W. Aiken, *Piloted Simulation of One-On-One Helicopter Air Combat at NOE Flight Levels*, USAAVSCOM Technical Report 85-A-2, NASA Ames Research Center, Moffett Field, California, April, 1985

Anil V. Phatak, Hien H Tran, *A3I Autopilot/Guidance Program Homing/Path Guidance with Turn-Straight-Turn Option* (Version TGUIDAP2) AMA Report 252-3, Mountain View, California, March, 1988

Andrew P. Lui, *A3I Phase III Views*, A3I Project, NASA Ames Research Center, Moffett Field, California, December, 1988

2.2 Information Documents

Silicon Graphics Inc., *IRIS-4D Series FORTRAN Programming Language*, Version 1.0, Mountain View, California, 1988

Silicon Graphics Inc., *IRIS GTX : A Technical Report*, Revision 2, Mountain View, California, 1988

Software Systems Inc., *MultiGen Reference Manual*, Version 1.0, San Jose, California, 1988

3.0 REQUIREMENTS AND DESIGN APPROACH

3.1 Requirements and Rationale

One of A3I's main goals is to replace the "man-in-the-loop" of ordinary manned simulations with models and principles of human performance in order to evaluate potential cockpit designs. Because such a large portion of aircrew tasks involve the actual "flying" of the vehicle, we need to adopt a representative model of helicopter flight dynamics, as well as a model of the guidance and navigation capabilities a pilot would use to employ the vehicle. Furthermore, because helicopter flight dynamics and controls are not commonly understood by the "typical" crew station designers (notional users of A3I), we felt it is beneficial to demonstrate visually the controls and how they affect the flight dynamics.

Because A3I's focus is on crew station design -- not helicopter aerodynamics or propulsion design, high levels of fidelity were not deemed necessary. However, because operational crew station design certainly cannot be achieved by neglecting the controls and tasks involved with "flying", a reasonable level of sophistication was necessary. Therefore, a flight dynamics/guidance model, which should be relatively easy to fly and yet exhibits all the major dynamic characteristics of a typical helicopter, is needed. The current Flight Dynamics/Guidance CSCI was developed by integrating TGUIDAP2/TMAN, written in Fortran by AMA, with the A3I's Views CSCI, because TGUIDAP2/TMAN alone does not render any graphics display capabilities. The current model represents rather generic helicopter dynamics, intended to be tailored by potential users (or replaced with a similar model) to match the performance of the vehicle under development.

TGUIDAP2/TMAN adopts the widely-used Euler approach to the helicopter orientation which involves two sets of right-handed orthogonal axes -- body axes and earth axes. Body axes consist of an axis system fixed in the helicopter, with the origin at its center of gravity and the x axis aligned with the fuselage reference line. Body axes thus move in space and rotate with the helicopter. Earth axes comprise a set of axes defined with respect to the earth, with the origin at a suitable point, the x axis pointing North, the y axis East, and the z axis down. Unlike body axes, earth axes are inertial axes. They are used as reference axes for position and attitude. Fig. 3.1 depicts the system of body axes used in TGUIDAP2/TMAN and the three rotational angles -- yaw, pitch, and roll.

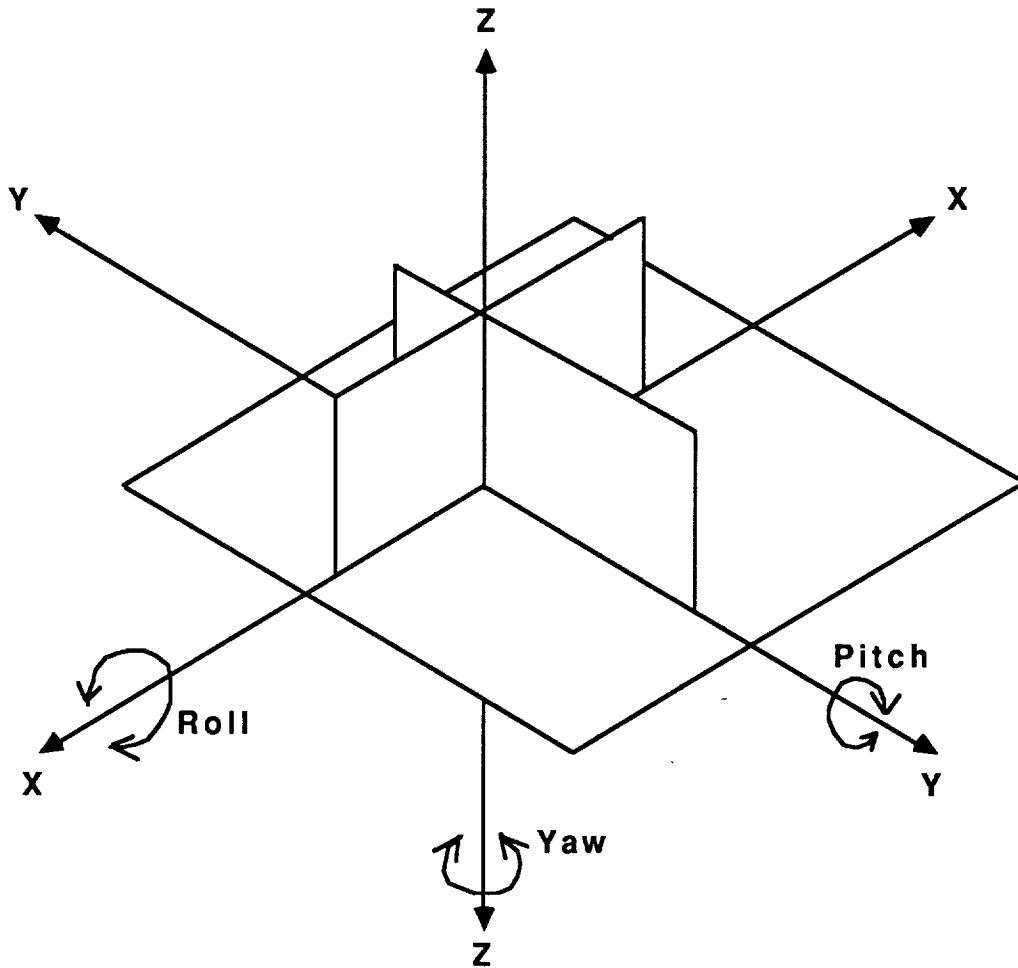


Fig. 3.1

The current implementation controls yaw, pitch, roll, and altitude through conventional maneuver, i.e., pedals, cyclic, and collective. The Four axis controllers, envisioned for future aircraft are not currently supported. The pedals are used to fix the attitude of the helicopter in rotation about the vertical axis, permitting the pilot to point the helicopter in any horizontal direction, namely, the yaw attitude. The cyclic is used for longitudinal and lateral control, namely, the pitch and roll attitude. The collective is used for altitude control of the helicopter in the vertical direction. The pilot's yaw, pitch, roll, and vertical control inputs are expressed in terms of percent of full scale and are limited in the range (-50%,50%).

In TGUDIAP2/TMAN, the coordinates x, y, and z are expressed in terms of feet, roll, pitch, and yaw in terms of degrees, and the translational and rotational velocity components in terms of ft/sec and rad/sec, respectively. The airspeed is expressed in terms of knots -- one knot is approximately 1.69 ft/sec. The Euler angles take the following range of values

roll : $(-\pi, \pi)$,
pitch : $(-\pi/2, \pi/2)$,

yaw : $(-\pi, \pi)$.

While the Flight Dynamics/Guidance CSCI used for the A3I Phase II simulation resided on a Symbolics 3675, Phase III efforts moved this CSCI to an IRIS/4D 70G. One of the reasons for this change was that the IRIS/4D 70G has a better Fortran environment than the Symbolics 3675. Due to the autonomous nature of the flight dynamics/guidance model, the aircraft traversed all the waypoints in Phase II simulation without interruption. The lack of interruptability makes it difficult for the simulated pilot to re-route the flight path when necessary, and thus imposes artificial constraints on the mission decomposition and simulation. It also lacked the capability for the pilot to directly alter the controls. Therefore, in spite of the fact that it provided flight dynamics information to other A3I CSCIs, it needed to be enhanced to interact with other CSCIs more flexibly. This became one of the guidelines for the Phase III flight dynamics/guidance development. In fact, new capabilities have been implemented on top of the AMA developed code which represent continuing efforts toward rendering a better flight dynamics/guidance model to meet evolving A3I requirements.

3.2 Hardware Environment

The Phase III flight dynamics/guidance development was originally done on a Silicon Graphics IRIS/4D 70G running UNIX System V with Berkeley extension BSD 4.3 and the 4Sight windowing system. This IRIS/4D 70G was then upgraded to 70GTX with parallel processing capability and the Phase III flight dynamics/guidance CSCI has been ported accordingly. For more information on the 70GTX, refer to the relevant document listed in Section 2.2.

3.3 Software Environment

The Flight Dynamics/Guidance CSCI consists of two CSCs, namely, TGUIDAP2 and TMAN. Prior to running TGUIDAP2/TMAN stand-alone, a set of waypoints has to be specified; normally, the specification is made by the designer during the mission planning phase using Symbolics. For each waypoint, five items have to be specified -- x, y, z, airspeed, and heading. The main routine of TGUIDAP2/TMAN has a do loop which iterates the following process until all the waypoints have been traversed: First, TGUIDAP2 is invoked to compute the control inputs based on the current position and the next waypoint. With these control inputs, TMAN integrates the three rotational equations for body-axis roll, pitch, and yaw accelerations to yield the body-axis angular rates, converts the body-axis angular rates to Euler angular rates, and integrates the Euler angular rates to obtain Euler angles. The elements of the body axis-earth axis transformation matrix T are then formed using the sines and cosines of the Euler angles. TMAN then transforms the body-axis longitudinal, lateral, and vertical forces to earth-axis forces using the transformation matrix T . Accelerations in the earth axis system are then calculated from the earth axis forces. Integrating these accelerations yields the velocities, which are in turn integrated to yield the translational displacements. It is necessary to transform the earth-referenced velocity components to the body axis velocity components because these will be used in the next cycle. The time step for all the integrations performed is a tick, which was set to one tenth of a second for Phase III simulation. The above solution sequence of the Euler approach is summarized as follows.

Body axis angular accelerations --> body axis angular rates --> Euler angular rates --> Euler angles
 --> body axis-earth axis transformation matrix --> body axis forces --> body axis accelerations -->
 body axis velocity components --> earth axis velocity components --> earth axis displacements

For more detail of the Euler approach, refer to the second reference listed in Section 2.1.

As mentioned earlier, the flight dynamics/guidance demonstration was implemented by integrating the Flight Dynamics/Guidance CSCI written in Fortran with the Views CSCI written in C. Modifications were made to the code of both CSCIs to pass data through interlanguage calls or common data blocks. The TGUIDAP2/TMAN main routine has been modified such that it is callable by Views, and it has been properly inserted in the infinite loop of the Views main function. Each time the modified TGUIDAP2/TMAN main routine is invoked, TGUIDAP2 and TMAN are driven sequentially as described above. The updated controls and flight dynamics are then passed through the C-Fortran interface to the Views graphics routines which update the scenario. Note that the Views right-handed orthogonal coordinate system does not coincide with the TGUIDAP2/TMAN earth system, as depicted in Fig. 3.2. Conversion of the aerodynamics data to meet the Views direction convention must be performed before the Views graphics routines update the scenario.

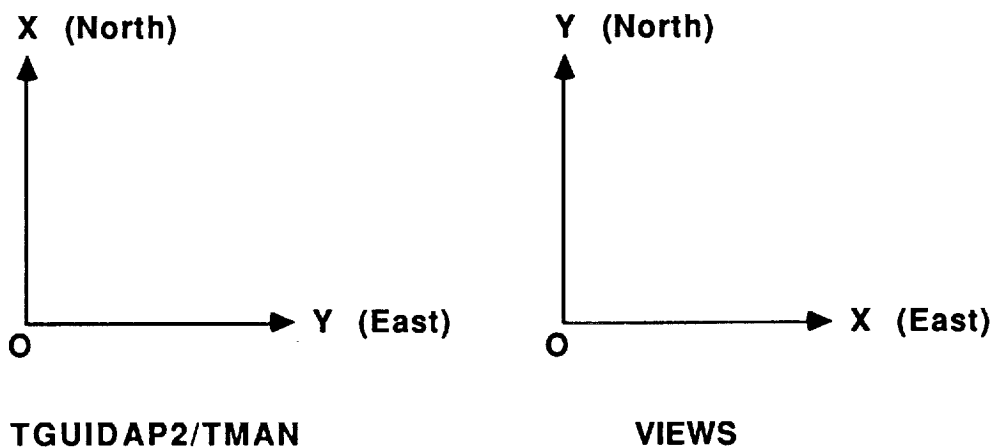


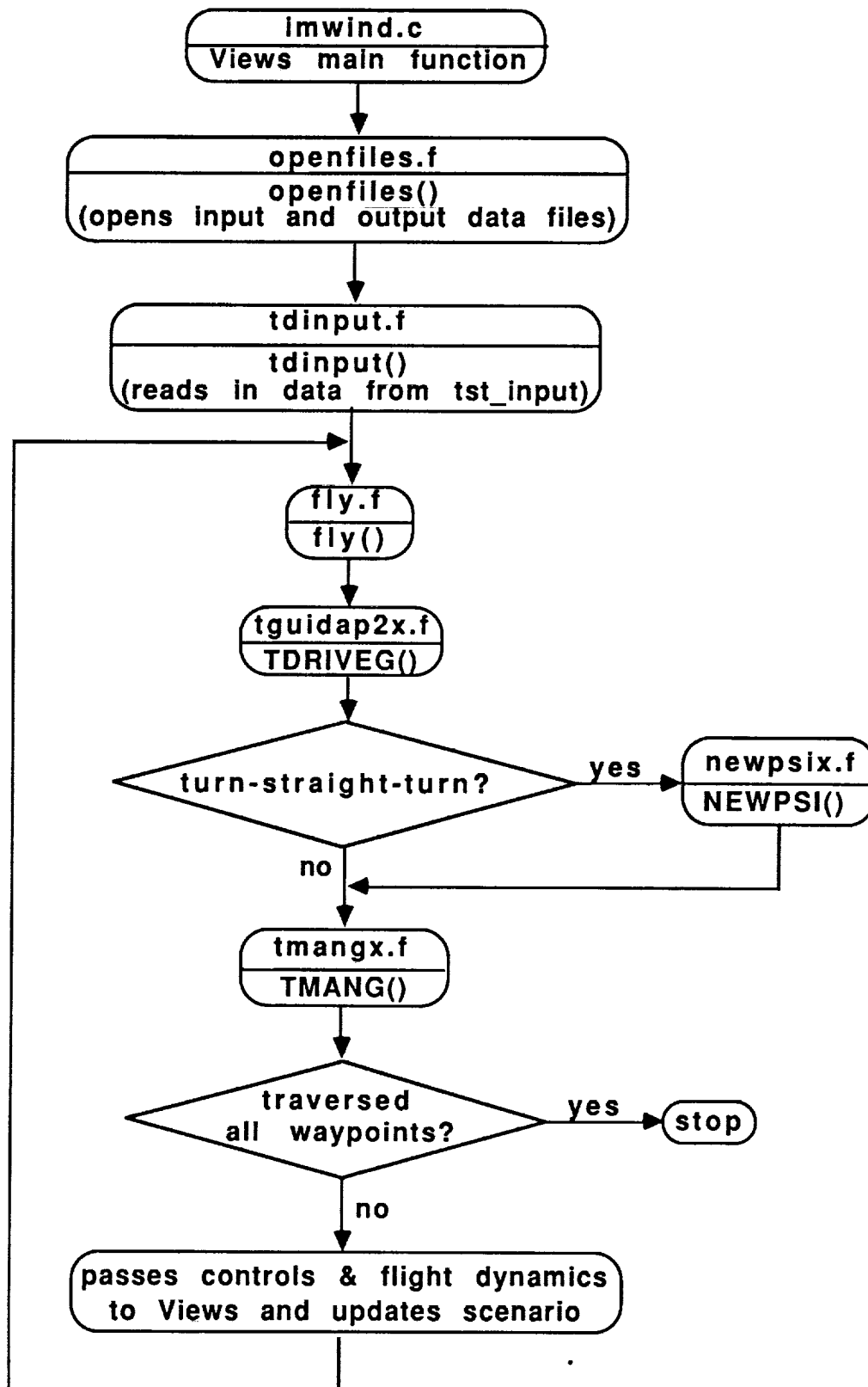
Fig. 3.2

4.0 DETAILED DESIGN DESCRIPTION

4.1 Organization

All the TGUIDAP2/TMAN Fortran files, ending with ".f", and Views C files, ending with ".c", reside on IRIS/4D 70G. The Views files can be found in the directory /usr/people/u/huimin/demo/mg3.0 and the TGUIDAP2/TMAN files in /usr/people/u/huimin/demo/mg3.0/heli.

A top level flow chart of the integrated TGUIDAP2/TMAN-Views package is depicted in Fig. 4.1



For more detailed flow charts and descriptions of Views and TGUIDAP2/TMAN refer to the documents listed in Section 2.1.

4.2 Unit Detailed Design

TGUIDAP2/TMAN has several units as shown in Fig. 4.1 and they are explained below. Only those Views files which have been modified are included in this section. For those unmodified Views files refer to *A3I Phase III Views*. Inputs and Outputs seen below stand for the input and output arguments to the functions.

4.2.1 openfiles.f

subroutine : openfiles()

Purpose: this subroutine opens input data files such as `tst_input` and altitudes, and output data files such as `tst_output`; each row of `tst_input` has four entries -- the first three entries contain values, and the last entry contains the variable names used in TGUIDAP2 and TMANG routines; the first two values are used by `tdinput.f` to determine an array element; a typical row of the data contained in `tst_input` is

```
1      168    0.100 DT2;
```

refer to `tst_input` for a complete list of array elements and the corresponding variables; the file "altitudes" specifies the altitude above terrain for each waypoint.

Inputs: none.

Outputs: none.

4.2.2 tdblkdatx.f

subroutine : BLOCK DATA

Purpose: `tdblkdatx.f` establishes the equivalence between the array elements determined by `tdinput.f` from the first two values specified in `tst_input` and the variables used in TGUIDAP2 and TMANG routines; an example is

```
EQUIVALENCE (DT2,A(168));
```

it also sets values for the variables, such as

```
DATA THDOT / 12. /.
```

Inputs: none.

Outputs: none.

4.2.3 tdinput.f

function : tdinput()

Purpose: `tdinput.f` reads in the four entries from each row of `tst_input` -- three values and one variable name; it processes the first value, which is used to identify the array name, and the second value, which is the array index, to yield the array element; it then assigns the third value to the array element; for example, the following row of data of `tst_input`

```
1      168    0.100 DT2
```

is processed by `tdinput.f` to yield

$A(168) = 0.1;$

DT2 is the variable name used in TGUIDAP2 and TMANG routines which stands for the time step of integration; DT2 is then set to 0.1 through the equivalence of DT2 and A(168) established in tdblkdatx.f.

Inputs: none.

Outputs: none.

4.2.4 fly.f

subroutine : fly(ithcycle)

Purpose: the executive program of TGUIDAP2/TMAN; it has been modified to be able to be called by the main function imwind.c of Views to drive TGUIDAP2/TMAN; it invokes TDRIVEG, which computes the required control movements and, in turn, invokes TMANG to compute new translational and rotational positions; when TDRIVEG is invoked, ithcycle is passed along as the only argument, and its value is altered to -1 by TDRIVEG if all waypoints have been traversed.

Inputs: ithcycle.

Outputs: ithcycle.

4.2.5 tguidap2x.f

subroutine : TDRIVEG(ithcycle)

Purpose: invoked by fly.f to compute the collective, pedal, and cyclic controls needed to steer the helicopter from the current position to the next waypoint; this routine does initialization for the first run; it gets the next waypoint if the helicopter is within the capture zone of the current waypoint; the computed controls are then passed to TMANG through a common data block; if all waypoints have been traversed, ithcycle is altered to -1 and returned to fly.f; otherwise, ithcycle remains unaltered.

Inputs: ithcycle.

Outputs: ithcycle.

4.2.6 newpsix.f

subroutine : newpsi (x1, y1, h1, r1, x4, y4, h4, r2, x2, y2, h2, x3, y3, h3, tr1, tr2, tr3, td1, td2, td3, d3, r3, kturn, xc1, yc1, xc2, yc2, s1, s2)

Purpose: invoked by tguidap2x.f only if the turn-straight-turn option is in use by setting NCASE to one in tst_input; the turn-straight-turn option has always been in use during the Phase III development and demonstration period because it renders high fidelity simulation; this routine computes all possible routes to go from the current position to the next waypoint based on horizontal position (x, y), heading, and airspeed associated with these two points, and chooses the shortest one.

Inputs: initial position, heading, and turn radius (x1, y1, h1, r1), and final position, heading, and turn radius (x4, y4, h4, r2); the headings h1 and h4 are expressed in radians and are in the range $(-\pi, \pi)$; r3 is zero in current application.

Outputs: same as above, only they are now updated.

4.2.7 tmangx.f

subroutine : TMANG ()

Purpose: invoked by TDRIVE to compute new position (x,y,z), roll, pitch, and yaw assuming that the collective, pedal, and cyclic controls computed by TDRIVEG are applied for one DT2, which is the integration time step defined in tst_input and is one tenth of a second.

Inputs: none.

Outputs: none.

4.2.8 imwind.c

Purpose: the main function of MultiGen; it has been modified to invoke openfiles.f to open input and output data files, tinput.f to read input data, fly.f to drive the flight dynamics/guidance CSCI, and finally the graphics routines to draw the scenario.

Inputs: none.

Outputs: none.

4.2.9 animate.c

Purpose: the animation control file; some flight dynamics/guidance pulldown menus have been added under the pulldown menu title "ANIMATE" such as Aero/Guidance Demo, Change Waypoint, Freeze Flight, Resume Flight, Freeze Collective, Unfreeze Collective, and Restart Aero Demo; clicking any of the these pulldown menus would invoke the proper function call to perform the desired functionality.

Inputs: none.

Outputs: none.

4.2.10 simdata.c

Purpose: this file has been modified so that it is able to receive the updated flight dynamics and control data -- x, y, z, roll, pitch, yaw, collective, pedal, and cyclic controls, etc., from the flight dynamics/guidance CSCI through a common data block, perform necessary conversion on the data to meet the MultiGen format, and update the MultiGen data structure for the helicopter.

Inputs: none.

Outputs: none.

4.2.11 path.c

Purpose: this file contains functions that provide the necessary interfaces for the user to change the next waypoint on-line when the pulldown menu "Change Waypoint" under the pulldown menu title "ANIMATE" is clicked, including the facilities to select a point from the terrain that specifies x, y,

and z, and a window to input the altitude above terrain, airspeed, and heading for the new waypoint; this file also contains routines to perform the necessary functionality when the pulldown menu "Draw flight path" under the title "ANIMATE" is clicked -- open the data file "waypoints", read waypoints, close the data file, and draw the flight path.

Inputs: none.

Outputs: none.

4.2.12 mgcom.c

Purpose: this file sets the correct the internal resolution to convert the flight dynamics/guidance data to conform with the MultiGen format.

Inputs: none.

Outputs: none.

4.2.13 mgedit.c

Purpose: this file contains the routines to initiate the procedure for the user to change next waypoint when the pulldown menu "Change Waypoint" is clicked.

Inputs: none.

Outputs: none.

4.2.14 checkerboard.2

Purpose: this binary file contains the data base of the scenario displayed in the world view including a grid with a horizontal ridge in the middle and a helicopter, and is the second argument of the command line to run the flight dynamics/guidance demo.

Inputs: none.

Outputs: none.

4.2.15 waypoints

Purpose: this file contains the information of all the waypoints which define the flight path; in this file there are seven items associated with each waypoint : waypoint number, x, y, z, altitude above terrain, speed, and heading; this is the data file that is opened when the pulldown menu "Draw flight path" is clicked.

Inputs: none.

Outputs: none.

4.2.16 animateflt.doc

Purpose: this file contains the ID numbers and the names for all objects; it is the file needed to open when the pulldown menu "Set up" under the title "ANIMATE" is clicked.

Inputs: none.

Outputs: none.

4.2.17 animate.menu

Purpose: this file contains all the names of the pulldown menus under the title "ANIMATE", including those related to the flight dynamics/guidance CSCI.

Inputs: none.

Outputs: none.

4.2.18 message.h

Purpose: this header file, used by MultiGen, contains a typedef struct to which four floating fields have been added to include the four controls -- collective, pedal, x-cyclic, and y-cyclic; this file constitutes part of the mechanism for passing data back and forth between TGUIDAP2/TMAN and MultiGen.

Inputs: none.

Outputs: none.

4.2.19 message.common

Purpose: this header file is the counterpart of message.h used by TGUIDAP2/TMAN; it has a common data block which contains the same set of information as that of message.h; this file constitutes part of the mechanism for passing data back and forth between TGUIDAP2/TMAN and MultiGen.

Inputs: none.

Outputs: none.

4.2.20 heli.h

Purpose: this header file, used by MultiGen, has a typedef struct, which contains the variables used in "Change waypoint", "Freeze collective", "Change climb rate", etc.; this file constitutes part of the mechanism for passing data back and forth between TGUIDAP2/TMAN and MultiGen.

Inputs: none.

Outputs: none.

4.2.21 heli.common

Purpose: this header file is the counterpart of heli.h used by the flight dynamics/guidance module; it has a common data block which contains the same set of information as that of heli.h; this file constitutes part of the mechanism for passing data back and forth between TGUIDAP2/TMAN and MultiGen.

Inputs: none.

Outputs: none.

5.0 NOTES

5.1 Miscellaneous

5.2 Limitations

Some of the new capabilities added on top of TGUIDAP2/TMAN work quite well, some of them need to be improved. Among them "Aero/Guidance Demo", "Freeze Flight", "Resume Flight", "Freeze Collective", and "Unfreeze Collective", and "Restart Aero Demo" work successfully, but "Change Waypoint" needs to be improved. At this point, the "Change Waypoint" capability works well only when the command is issued way before the helicopter reaches the next waypoint. This will be improved to such an extent that the command of changing the next waypoint can be issued any time before the helicopter gets in the capture zone.

5.3 Future Directions

A higher fidelity modeling within TGUIDAP2/TMAN is envisioned as necessary to meet evolving A3I requirements. The goal is to make TGUIDAP2/TMAN more controllable by the pilot model in the manner that pilots usually fly. The wind effect may be incorporated into TGUIDA2/TMAN in the future development. The current requirement to specify heading, altitude, etc. at each waypoint may need to be relaxed because the complete set of such data is not known. Good dynamics model for land and sea vehicles need to be developed to expand the application potential for the A3I workstation.

6.0 USERS GUIDE

The following describes how to set up the flight dynamics/guidance demo.

- (1) After logging on to 4D console or through the dummy terminal hooked to 4D, type "cd /usr/people/u/huimin/demo/mg3.0/Mg/usr/mg" to change to the directory where the flight dynamics/guidance demo executable and the necessary files reside.
- (2) Do "mgflt.share checkerboard.2"; the pulldown menu bar, the edit control window, the icon editors, the coordinate window, and a window with name "checkerboard.2" and with a small piece of terrain appear on the console.
- (3) Click the option "ANIMATE" on the menu bar, drag the mouse down to the menu "Set up", and then release the mouse.
- (4) Hit the carriage return when a temporary window with the default file name "animateflt.doc" shows up on the screen; the temporary window then goes away.
- (5) Another temporary window with all the object names and the assigned group ID's shows up on the screen; click "DONE" in the temporary window; the temporary window then goes away.

- (6) Click the menu option "SELECT" on the menu bar, drag the mouse down to the pulldown menu "From ID", then release the mouse; a temporary window waiting for user to enter the object name shows up on the screen.
- (7) Type G4 to the temporary window and hit the carriage return; the temporary window goes away.
- (8) Hit the dot key to center the piece of terrain in the "checkerboard.2" window.
- (9) Click the menu option "SELECT" on the menu bar, drag the mouse down to the pulldown menu "Push Prio", and release the mouse.
- (10) Click menu title "SELECT" on the menu bar, drag down to "Deselect all", and release mouse.
- (11) Click the menu option "ANIMATE" on the menu bar, drag the mouse down to "Moving Camera", then release the mouse; a blue window shows up.
- (12) Click "DONE" in the blue window; the blue window then goes away and a window representing the pilot view shows up at the lower left corner.
- (13) Repeat step (11).
- (14) Click the last item of the blue window and click "DONE"; the blue window goes away; a window representing the observer's view shows up.
- (15) Arrange the pilot view and the observer's view in any way you like.
- (16) Click the menu option "ANIMATE" on the menu bar, drag the mouse down to "Draw Flight Path", then release the mouse; a temporary window shows up prompting for the data file name; type "waypoints" to the window and hit the return key; the flight path will be drawn in the "checkerboard.2" window.
- (17) Click the menu option "ANIMATE" on the menu bar, drag the mouse down to "Control Display", then release the mouse; this step is to set up the three control windows -- cyclic, pedal, and collective; but they will not get drawn until the flight dynamics/guidance demo starts.
- (18) Now the set up is done; you can go ahead start the flight dynamics/guidance demo by clicking the menu option "ANIMATE", dragging the mouse down to "Aero/Guidance Demo", and releasing the mouse.

The pulldown menu "Freeze Flight" is provided for the user to freeze the demonstration for as long as needed. To resume, click "Resume Flight". Clicking "Restart Aero Demo" will start the demo from scratch. "Freeze Collective" allows the user to freeze the collective control for as long as he wants, and the helicopter will fly with the frozen collective control until the user clicks "Unfreeze Collective".

Annex G

Army-NASA Aircrew/Aircraft Integration Program

A³I

**Software Detailed Design Document:
Communications**

prepared by

Alex Chiu

December 1988

Table of Contents

1.0	INTRODUCTION.....	G-1
1.1	Identification.....	G-1
1.2	Scope.....	G-1
1.3	Purpose	G-1
2.0	RELATED DOCUMENTATION	G-1
2.1	Applicable Documents.....	G-1
2.2	Information Documents	G-1
3.0	REQUIREMENTS AND DESIGN APPROACH	G-1
3.1	Requirements and Rationale	G-2
3.2	Hardware Environment	G-2
3.3	Software Environment	G-3
4.0	DETAILED DESIGN DESCRIPTION	G-3
4.1	Organization.....	G-3
4.2	Unit Detailed Design.....	G-4
4.2.1	multi-stream.lisp	G-4
4.2.2	multi_stream.c.....	G-4
5.0	NOTES.....	G-4
5.1	Miscellaneous	G-4
5.2	Limitations.....	G-5
5.3	Future Directions.....	G-5
6.0	USERS GUIDE.....	G-5

1.0 INTRODUCTION

1.1 Identification

This document establishes the requirements and detailed design of the Communication Computer Software Configuration Item (CSCI), which forms a part of the A3I Computer Program System. Descriptions of the detailed processing requirements, structure, I/O, and control are provided for each lower level Computer Software Component (CSC), unit, or function contained within the CSCI.

1.2 Scope

This document is focused on the Phase III development of the A3I communication CSCI -- particularly the development of a set of TCP/IP-based communication application routines for bi-directional data transmission across the Ethernet between Symbolics 3675 and IRIS/4D, and between IRIS/4D and IRIS/2500T. It is assumed that the reader is familiar with UNIX, C, Genera, Symbolics Common Lisp, and the basic concept of local area network communication.

1.3 Purpose

The purpose of the A3I Communication CSCI is to provide a mechanism for the A3I CSCIs running on different operating system workstations -- four Symbolics workstations and four IRIS workstations to transmit data bi-directionally. The data may be in the form of characters, short or long integers, and floating points. An IBM/AT compatible is also available which will serve as the A3I communication control center to initiate the set up of all the necessary byte streams when the A3I simulation starts, and also as the A3I diagnostic center with a LANalyzer board installed in it. Previous development phases provided a means of one-way data transmission from Symbolics 3675 to IRIS 2500T which is inadequate for A3I's purposes. This CSCI has been enhanced to allow bi-directional data transmission between Symbolics 3675 and IRIS/4D, and between IRIS/4D and IRIS/2500T.

2.0 RELATED DOCUMENTATION

2.1 Applicable Documents

Tanenbaum, A. S. Silicon Graphics Inc., "Computer Networks", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981

Symbolics Genera 7.2 Manual, Vol. 5, "Streams, Files, and I/O"

Symbolics Genera 7.2 Manual, Vol. 9, "Networks"

2.2 Information Documents

3.0 REQUIREMENTS AND DESIGN APPROACH

3.1 Requirements and Rationale

Efficiency, modularity, reliability, flexibility, and friendly interface are always the guideline for the development of the communication CSCI.

The current A3I configuration consists of four Genera-based Symbolics workstations (3675, two 3640s, 3620), four UNIX-based IRIS workstations (4D/70GTX, 3120, 2500T, Personal IRIS) with Berkeley extension BSD 4.3, and a DOS-based IBM/AT. The task of providing facilities for such a configuration to transmit data bi-directionally can be decomposed into five subtasks, based upon the operating systems the workstations run. These five subtasks are to provide facilities for data transmission (1) between Genera and Genera, (2) between UNIX and UNIX, (3) between Genera and UNIX, (4) between DOS and UNIX, and (5) between DOS and Genera.

The emphasis of the Phase III communication CSCI development was on subtask (3) such that characters, short and long integers, and floating points could be sent back and forth between Symbolics 3675 and IRIS/4D/70GTX.

High transmission rate, high reliability, and ease in upgrading are the main reasons that Ethernet is employed as the hardware cable to transmit data. Among the available protocols, TCP/IP is employed because it provides accurate transmission control protocol, is well accepted by the communication community, and is available on all Symbolics and IRIS workstations.

To achieve the transmission most effectively, short or long integers and floating points are transmitted in binary format, while character strings in ASCII format.

3.2 Hardware Environment

Each workstation is a node in the network. Ethernet is the primary hardware used to transmit data among the nodes. All nodes share the same channel and will hold their messages when there are ongoing transmissions until the channel is clear. Listed below are some primary attributes of the Ethernet.

Topology : Bus.

Medium : Shielded coaxial cable.

Data Rate : 10 million bits per second.

Maximum Separation of Nodes : 2.8 kilometers (about 1.7 miles).

Maximum Number of Nodes : 1024.

Network Control : Multi-access.

Access Control : CSMA/CS.

Allocation : 64 to 1518 bytes per packet.

Connecting the Symbolics (or IRIS) to an Ethernet local area network requires following hardware:

- An Ethernet transceiver to attach to the Ethernet.
- A drop cable to connect the Symbolics (or IRIS) cabinet to the Ethernet transceiver.
- An Ethernet board in Symbolics (or IRIS) workstation.

The IRIS/4D 70GTX runs UNIX V with Berkeley extension and 4Sight windowing system while the Symbolics 3675 runs Genera 7.2.

3.3 Software Environment

The IRIS communication software is written in C and the Symbolics communication software in Lisp, in client-server fashion with the IRIS/4D 70GTX being the server and the Symbolics 3675 the client. These sets of software not only set up the network at the beginning of simulation but also handle sending and receiving data. Caution must be exercised because the bytes get swapped during the transmission of integers, short or long, and floating points between these two different operating system workstations. To ensure that both the Symbolics 3675 and the IRIS/4D 70GTX receive correct data, action must be taken to overcome the byte swap problem. This is done on the Symbolics 3675 because functions are available there to perform the swapping of bytes. Before the Symbolics 3675 sends data across the network and after it receives data from the network -- (1) for short integers, swap the higher byte with the lower byte, and (2) for long integers and floating points, swap the first high byte with the first low byte, and the second high byte with the second low byte. The byte swap can also be overcome by sending integers and floating points in ASCII format, but it may need more bytes to send the same integer or floating point than in binary format.

4.0 DETAILED DESIGN DESCRIPTION

4.1 Organization

The data passing between the 3675 Symbolics and the IRIS/4D 70GTX is depicted in the following figure.

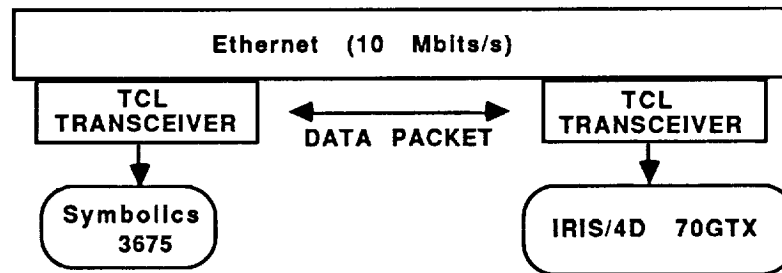


Fig. 4.1

The following describes the most significant piece of communication software developed in Phase III. It provides a link to integrate the Cognitive Pilot CSCI, which runs on the Symbolics 3675, TGUIDAP2/TMAN (the Flight Dynamics/Guidance CSCI), and JACK (the Mannequin CSCI), both run on IRIS/4D 70 GTX with JACK in foreground and TGUIDAP2/TMAN in background.

An integration template has been implemented to validate this piece of communication software. Appropriate communication C code has been properly inserted in JACK code to enable him to accept the request for connection from the Symbolics 3675 when JACK is run with the communication option, and to send/receive messages to/from the Symbolics 3675 across the network. A pseudo pilot model has also been used to replace the Cognitive Pilot CSCI in sending/receiving messages to/from IRIS/4D 70GTX.

When this template starts running, it sets up the client-server network. The pseudo pilot model then sends an activity script to JACK to execute. The pseudo pilot model receives the completion signal sent from JACK after JACK is done with the script. Then TGUIDAP2/TMAN sends the flight dynamics and control parameters to the pseudo pilot model. After the pseudo pilot model receives the flight dynamics and control parameters it sends a new script to JACK to start the next cycle. The network closes after the process continues for a couple of hundred cycles.

4.2 Unit Detailed Design

Described in this section includes the detailed design for both the Symbolics 3675 and the IRIS/4D 70GTX communication software.

4.2.1 multi-stream.lisp

Purpose : this file is the pseudo pilot model running on the Symbolics 3675; it defines protocols and eight byte streams -- four for the pseudo pilot model to communicate with JACK, and another four for the pseudo pilot model to communicate with TGUIDAP2/TMAN; it handles hand shaking and transmission of various types of data over appropriate byte streams, sends activity scripts to JACK across the Ethernet; it receives completion signal from JACK, and flight dynamics and control parameters from TGUIDAP2/TMAN; it closes the network after a number of cycles. (to have a proper close of the network the client should always close first.)

Inputs: none.

Outputs: none.

4.2.2 multi_stream.c

Purpose : the file contains functions to set up four streams for the pseudo pilot model to communicate with TGUIDAP2/TMAN and another four streams for the pseudo pilot model to communicate with JACK by making a series of system calls; it also contains functions for JACK to receive activity scripts from the pseudo pilot model across the Ethernet, for JACK to send a message to the pseudo pilot model to signal the completion of the script, and for TGUIDAP2/TMAN to send flight dynamics and control parameters to the pseudo pilot model.

Inputs: none.

Outputs: none.

5.0 NOTES

5.1 Miscellaneous

In communication network regime, hand shaking is achieved when the client makes a request for connection and the server accepts the request. To set up a network, it is common practice to bring up the server (IRIS/4D 70GTX in our case) to the "listen" stage, waiting for the client to issue the

request for connection. The A3I network has been implemented in such a way that if the user forgets to first bring the server up it will remind the user to do so.

5.2 Limitations

When the communication is over it is advised that the the client (the Symbolics 3675 in our case) should be the one who initiates the close of the network. Otherwise, an improper close of the network will result, i.e., the port numbers allocated will not be properly released, and any future request for connection using the same set of port numbers will not succeed unless the system is rebooted.

5.3 Future Directions

- (1) Incorporate the IBM/AT compatible in the A3I simulation as the A3I communication debugging center by installing in it the LANalyzer. It serves as the A3I communication control center which initiates the set up and close of the network.
- (2) Expand the A3I network to include new workstations.

6.0 USERS GUIDE

The following describes the general procedure to set up the communication network for the Symbolics 3675 and the IRIS/4D 70GTX to run the integration template which includes the pseudo pilot model, JACK, and TGUIDAP2/TMAN.

- (1) After logging on to the IRIS/4D 70GTX (the server) through the dummy terminal hooked to it, type "cd /usr/people/u/huimin/b4tape/comm_back/s4d" to change to the directory where the communication executable resides.
- (2) Do "multi-stream" to bring the server to the "listen" stage; a message then pops up signaling that the server is ready to accept a request for connection.
- (3) Log on to the Symbolics 3675 (the client), and run the executable "establish-network" by typing (user::establish-network) and hitting the carriage return.
- (4) Click the "yes" icon when the window with the question "Is Coral ready?" shows up.
- (5) Click the "yes" icon when another temporary window shows up on the Symbolics 3675 asking if Coral is ready to accept the request for opening a byte stream to transmit characters.
- (6) Click the "yes" icon when another temporary window shows up on the Symbolics 3675 asking if Coral is ready to accept the request for opening a byte stream to transmit short integers.
- (7) Click the "yes" icon when another temporary window shows up on the Symbolics 3675 asking if Coral is ready to accept the request for opening a byte stream to transmit long integers.
- (8) Click the "yes" icon when another temporary window shows up on the Symbolics 3675 asking if Coral is ready to accept the request for opening a byte stream to transmit floating points.

At this point, the simulation proceeds, as described in Section 4.1, and terminates automatically after a couple of hundred cycles.

Although items (5) through (8) seem to be redundant, they are implemented to provide friendly user interface.



Report Documentation Page

1. Report No. NASA CR-177557		2. Government Accession No.		3. Recipient's Catalog No.	
4. Title and Subtitle Army-NASA Aircrew/Aircraft Integration Program (A ³ I) Software Detailed Design Document: Phase III				5. Report Date June 1990	
				6. Performing Organization Code	
7. Author(s) Carolyn Banda, Alex Chiu, Gretchen Helms, TehMing Hsieh, Andrew Lui, Jerry Murray, and Renuka Shankar				8. Performing Organization Report No. A-90197	
				10. Work Unit No. 505-61	
9. Performing Organization Name and Address Sterling Federal Systems, Inc. 1121 San Antonio Road Palo Alto, CA 94303-4380				11. Contract or Grant No. NAS2-11555	
				13. Type of Report and Period Covered Contractor Report	
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546-0001				14. Sponsoring Agency Code	
15. Supplementary Notes Point of Contact: Robert A. Carlson, Ames Research Center, MS 233-15 Moffett Field, CA 94035-1000 (415) 604-6036 or FTS 464-6036					
16. Abstract This report details the capabilities and design approach of the MIDAS (Man-machine Integration Design and Analysis System) computer-aided engineering (CAE) workstation under development by the Army-NASA Aircrew/Aircraft Integration (A ³ I) Program. This workstation uses graphic, symbolic, and numeric prototyping tools and human performance models as part of an integrated design/analysis environment for crewstation human engineering. Developed incrementally, the requirements and design for Phase III (Dec. 87-Jun 89) are described. Software tools/models developed or significantly modified during this phase included: 1) an interactive 3-D graphic cockpit design editor; 2) multiple-perspective graphic "views" to observe simulation scenarios; 3) symbolic methods to model the mission decomposition, equipment functions, pilot tasking and loading, as well as control the simulation; 4) a 3-D dynamic anthropometric model; 5) an inter-machine communications package; and 6) a training assessment component. These components were successfully used during Phase III to demonstrate the complex interactions and human engineering findings involved with a proposed cockpit communications design change in a simulated AH-64A Apache helicopter/mission that maps to empirical data from a similar study and AH-1 Cobra flight test.					
17. Key Words (Suggested by Author(s)) Computer-aided engineering, Human performance modelling, Crewstation design, Man-machine interface, Human factors engineering				18. Distribution Statement Unclassified-Unlimited Subject Category - 54	
19. Security Classif. (of this report) Unclassified		20. Security Classif. (of this page) Unclassified		21. No. of Pages 318	
				22. Price A14	

